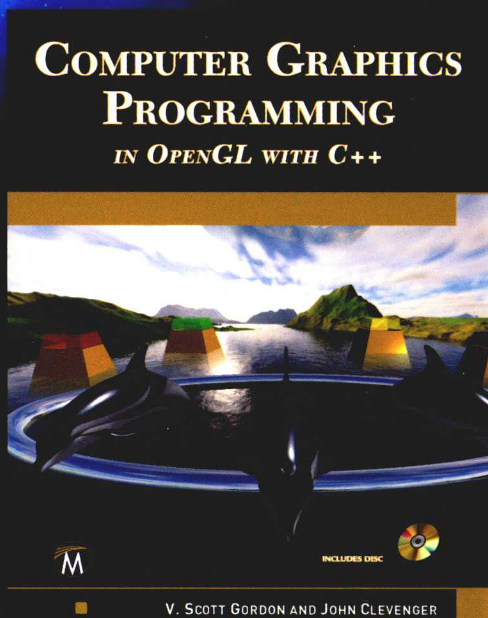
国外著名高等院校
信息科学与技术优秀教材

MERCURY
LEARNING AND INFORMATION LLC

异步图书
www.epubit.com

计算机图形学编程
（使用OpenGL和C++）

[美] V.斯科特·戈登（V. Scott Gordon）
[美] 约翰·克莱维吉（John Clevenger） 著
魏广程 沈瞳 译




中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

计算机图形学编程（使用 OpenGL 和 C++）

Computer Graphics Programming in OpenGL with C++

本书使用 OpenGL 和 C++，教授现代 3D 图形着色器编程。本书从图形编程的基础和准备工作开始，介绍了着色器的各个阶段，包括建模、光照、纹理等基础知识，以及曲面细分、柔和阴影、生成逼真的材质和环境等高级技术实现。

本书为读者提供丰富的学习素材，包括书中使用的所有源代码、模型、纹理、天空盒以及法线贴图，既适合作为高等院校计算机相关专业的计算机图形编程课程的教材或辅导书，也适合对计算机图形编程感兴趣的读者自学。

本书具有以下特色：

- 覆盖 C++ 中的现代 OpenGL 4.0+ 着色器编程；
- 使用可运行的代码示例讲解所有技术，提供完整的源代码以及详细的讲解。
- 详细讲解每个 GLSL 可编程管线阶段（顶点阶段、曲面细分阶段、几何阶段以及片段阶段）。
- 研究有关建模、光照、阴影（包括柔和阴影）、地形以及 3D 材质（例如木材和大理石）的实例。
- 介绍现代开发工具（如 NVIDIA Nsight 调试器），以及如何用其优化代码、提高性能。
- 提供书中使用的所有源代码、模型、图表、纹理、天空盒、天空穹顶、高度贴图和法线贴图。
- 本书为授课教师提供 PPT、习题解答、课程大纲等教学辅助资源，请通过 contact@epubit.com.cn 联系获取。

异步社区 www.epubit.com



异步社区 www.epubit.com
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn



ISBN 978-7-115-52128-6
定价：69.00 元

分类建议：计算机 / 程序设计
人民邮电出版社网址：www.ptpress.com.cn

国外著名高等院校
信息科学与技术优秀教材

计算机图形学编程
（使用OpenGL和C++）

[美] V.斯科特 · 戈登（V. Scott Gordon）
[美] 约翰 · 克莱维吉（John Clevenger） 著
魏广程 沈瞳 译

人民邮电出版社
北京

图书在版编目（CIP）数据

计算机图形学编程 : 使用OpenGL和C++ / (美) V.斯科特·戈登 (V. Scott Gordon) , (美) 约翰·克莱维吉 (John Clevenger) 著 ; 魏广程, 沈瞳译. -- 北京 : 人民邮电出版社, 2020.2
国外著名高等院校信息科学与技术优秀教材
ISBN 978-7-115-52128-6

Ⅰ. ①计… Ⅱ. ①V… ②约… ③魏… ④沈… Ⅲ. ①计算机图形学—高等学校—教材②C++语言—程序设计—高等学校—教材 Ⅳ. ①TP391.411②TP312.8

中国版本图书馆CIP数据核字(2019)第218712号

版权声明

◆ 著 [美] V.斯科特·戈登（V. Scott Gordon）
[美]约翰·克莱维吉（John Clevenger）
译 魏广程 沈瞳
责任编辑 陈冀康
责任印制 王 郁 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆ 开本：787×1092 1/16
印张：16.75 彩插：4
字数：397 千字 2020 年 2 月第 1 版
印数：1 – 2 400 册 2020 年 2 月北京第 1 次印刷
著作权合同登记号 图字：01-2018-7739 号

定价：69.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

本书以 C++和 OpenGL 作为工具，教授计算机图形学编程。全书共 14 章和 3 个附录。首先从图形编程的基础和准备工作开始，依次介绍了 OpenGL 图像管线、图形编程数学基础、管理 3D 图形数据、纹理贴图、3D 模型、光照、阴影、天空和背景、增强表面细节、参数曲面、曲面细分、几何着色器，以及其他相关的图形编程技术。附录分别介绍了 Windows、macOS 平台上的安装设置，以及 Nsight 图形调试器的应用。本书每章最后配备了不同形式的习题，供读者巩固所学知识。

本书适合作为高等院校计算机科学专业的计算机图形编程课程的教材或辅导书，也适合对计算机图形编程感兴趣的读者自学。

前　言

本书的主要目标是用作计算机科学专业本科 OpenGL 3D 图形编程相关课程的教材。同时，我们也付出了很大的努力，让本书成为一本无须配合课程使用的自学教材。在以这两者为目标的前提下，我们尽力将内容解释得简单而清晰。本书中的所有代码示例都已经尽可能地简化，同时没有破坏其完整性，以便读者可以直接运行。

我们期望本书与众不同的一点是，新手（刚接触 3D 图形编程的人）更容易学习。关于这个主题的学习资料从来都不匮乏，恰恰相反，很多新手刚入门的时候，相关的资料就扑面而来了。我们刚接触 3D 图形编程的时候，就期望能遇见这样的教材——一步步解释基础概念，循序渐进并有序地梳理进阶概念，因此我们也尝试将本书编写成这样的教材。我们曾想将本书命名为 *shader programming made easy*（《轻松学着色器编程》），虽然我们并不认为有什么方法能真的让着色器编程变得“轻松”，但我们希望本书能够帮助你尽可能地达成这个目标。

本书使用 C++进行 OpenGL 编程教学。使用 C++学习图形编程有以下几个好处。

- 由于 OpenGL 的原生语言是 C，因此 C++程序可以直接进行 OpenGL 函数调用。
- C++编写的 OpenGL 应用程序通常有着很好的性能。
- C++提供 C 语言所没有的现代编程结构（类、多态等）。
- 在 OpenGL 社区中，C++是一个热门选项。许多 OpenGL 的教学资源有 C++版本。

值得一提的是，OpenGL 也存在着与其他语言绑定。常见的有 Java、C#、Python 等，但本书仅关注 C++。

本书与众不同的另一点是它有一个 Java 版，英文书名是 *Computer Graphics Programming in OpenGL with Java*。这两本书是按同样的节奏组织的，它们使用相同类型的章节编号、主题、图表、习题和讲解方式，其代码组织方式也尽可能地相似。诚然，使用 C++或 Java 编程肯定有着相当大的差异。尽管如此，我们相信这两本书提供了几乎相同的学习路径，甚至可以让选修同一门课的学生使用不同的语言版本作为教材。

需要说明的一点是，OpenGL 有着不同的版本（稍后简述）和不同的变体。例如，在标准 OpenGL（也称桌面 OpenGL）之外，还有一个变体叫作 OpenGL ES。它是为嵌入式系统（Embedded System）的开发而定制的（因此称为 ES)。“嵌入式系统”包括手机、游戏主机、汽车和工业控制系统之类的设备。OpenGL ES 的大部分内容是标准 OpenGL 的子集，删除了嵌入式系统通常用不到的很多操作。OpenGL ES 还增加了一些功能，通常是特定目标环境下的特定功能。本书侧重于标准 OpenGL。

OpenGL 的另一种变体称为 WebGL。WebGL 基于 OpenGL ES，它的设计目标是支持在浏览器中运行 OpenGL。WebGL 允许应用程序通过 JavaScript 进行 OpenGL ES 操作调用，从而简单地将 OpenGL 图形嵌入标准 HTML（Web）文档中。大多数现代 Web 浏览器，包括 Apple Safari、Google Chrome、Microsoft Internet Explorer、Mozilla Firefox 和 Opera，支

持 WebGL。由于 Web 编程超出了本书的讨论范围，因此本书不会涵盖 WebGL。不过，由于 WebGL 基于 OpenGL ES，而 OpenGL ES 又基于标准 OpenGL，因此本书所涵盖的大部分内容可以直接迁移到这些 OpenGL 变体的学习中去。

3D 图形编程这个主题通常让人想起精美而宏大的画面。事实上，许多相关热门教材中充满了令人惊叹的场景，很大程度上是为了吸引读者翻阅他们的图库。虽然我们认同这些图例的激励作用，但我们的目标是教学而非令人惊叹。本书中的图像仅仅是示例程序的输出。由于本书只是入门教程，其渲染的场景应该无法让专家侧目。然而，本书呈现的技术确实是构成当今这些炫目 3D 效果的基础。

我们没有尝试写一本“OpenGL 参考大全”，因此，本书所涵盖的 OpenGL 部分只是其所有功能中的一小部分。我们的目标是以 OpenGL 作为基础工具，教授基于现代着色器的 3D 图形编程，并为读者提供足够深入的理解，以供进一步研究。

目标读者

本书的主要目标读者是计算机科学专业的学生（可以是本科在读学生），其实任何想要学习计算机科学相关知识的人也适合阅读本书。因此，我们假设读者有扎实的面向对象编程基础，至少相当于计算机科学专业大二或大三学生的水平。

还有一些本书没有涵盖的内容，因为我们假设读者已经掌握了足够的背景知识，包括：

- C++及其常用库，如标准模板库（Standard Template Library）；
- 熟悉集成开发环境（Integrated Development Environment，IDE），如 Visual Studio；
- 事件驱动编程概念；
- 基础矩阵代数、三角函数；
- 了解颜色模型，如 RGB、RGBA 等。

希望本书的潜在受众能够因为对其 Java 版的喜爱而进一步支持本书。正如前面所说的，我们期望看到这样一种情景——学生在同一门课中可以自由选择使用 C++或 Java 版本的教材。由于这两本书按同样的节奏对教学内容进行组织编排，因此我们认为可以尝试以这种开放的方式来开展课程教学和学习。

如何使用本书

本书从内容安排上适合从前往后阅读，即后面各章中的内容经常依赖于前面各章中所讲的内容。因此，在各章中来回跳跃地选择性阅读可能并不适合本书，读者最好逐章顺序阅读。

本书同时可以作为实用的动手指南。由于已经有许多其他偏理论的学习材料，因此读者应该将本书作为一本“练习册”，通过一边参考本书一边自己动手编程来理解基础概念。虽然我们为所有的示例提供了代码，但是想要真正理解这些概念，还是得自己动手“实现”这些代码——通过编程来搭建你自己的 3D 场景。

本书在第 2 章到第 14 章的最后都留给读者一些习题。有的题比较简单，仅仅需要对提供的代码进行简单改动就可以解决。那些标记为“项目”的习题，则需要读者花费更多的时间来解答，因为可能需要编写大量代码或者使用多个示例中用到的技术。少数标记为“研

究”的习题，则在本书中并没有提供解题需要学习的细节知识，我们鼓励读者进行自主学习并解答。

OpenGL 调用通常会有很长的参数列表。在撰写本书时，两位作者在每种情况下都会讨论是否要描述所有的参数。最终我们决定在最初的部分描述所有参数。随着主题深入，我们避免在每次 OpenGL 调用中陷入细枝末节的描述过程（因为调用的次数很多），以防读者失去对全局的理解。因此，在浏览示例时，读者需要在手边准备 OpenGL 和所使用的各种库的参考资料。

为此，我们建议结合一些优秀的在线资源使用本书。OpenGL 的文档是绝对必要的。有关各种命令的详细信息，可以利用搜索引擎，或访问 OpenGL 的官方网站获取。

我们的示例中用到了称作 GLM 的数学库。在安装 GLM（见附录）后，读者应该找到其在线文档并将其加入书签。

本书中经常用到的另一个库是 SOIL2，用于读取和处理纹理图像文件，读者可能也需要定期查阅它的文档。SOIL2 没有中心化的文档资源，但读者通过 Web 搜索可以找到一些例子。

还有许多关于 3D 图形编程的图书，我们建议与本书并行阅读（例如要解决各章后的“研究”问题）。以下是我们经常提到的 5 本。

- [SW15] Sellers et al. *OpenGL SuperBible.*
- [KS16] Kessenich et al. *OpenGL Programming Guide.*
- [WO13] Wolff, *OpenGL 4 Shading Language Cookbook.*
- [AS14] Angel and Shreiner, *Interactive Computer Graphics.*
- [LU16] Luna, *Introduction to 3D Game Programming with DirectX 12.*

配套资源

本书提供随书的配套资源供读者下载，其内容有：

- 书中所有 C++ / OpenGL 程序和相关的实用类文件以及 GLSL 着色器代码；
- 各种程序和示例中使用的模型和纹理文件；
- 用于制作天空和地平线的天空顶和立方体贴图图像文件；
- 用于照明和表面细节效果的法线贴图和高度贴图；
- 本书中所有的图表以图像文件形式提供。

上述文件也可以通过访问异步社区（www.epubit.com）上的本书页面获取。

教师辅助

我们鼓励大学或学院的教师获取本书的教师辅助包，其中包含以下附加项：

- 一套完整的 PowerPoint 幻灯片，涵盖本书中的所有主题；
- 本书中大多数章末习题的答案和所需代码；
- 基于本书的课程大纲示例；
- 每章用于讲解材料的额外内容。

教师辅助包可以通过联系出版商获取：contact@epubit.com.cn。

致谢

本书中的许多内容是基于我们之前出版的 *Computer Graphics Programming in OpenGL with Java*。我们需要感谢许多帮助我们完成上一本书的人，他们继续为本书的编写提供了帮助。Java 版早期的草稿被用于加州州立大学萨克拉门托分校的 CSc-155（高级计算机图形编程）课程中，得到了学生的指正，他们还给出了修改建议（包括代码）。两位作者要特别感谢 Mitchell Brannan、Tiffany Chiapuzio-Wong、Samson Chua、Anthony Doan、Kian Faroughi、Cody Jackson、John Johnston、Zeeshan Khaliq、Raymond Rivera、Oscar Solorzano、Darren Takemoto、Jon Tinney、James Womack 及 Victor Zepeda 的建议。

我们也从许多教师那里得到了很好的反馈，他们采用 *Computer Graphics Programming in OpenGL with Java* 作为课程教材，同时向我们分享了他们的教学经验。塔尔萨大学的 Mauricio Papa 博士和我们进行了几次对我们非常有帮助的邮件沟通。Sean McCrory 对光照（第 7 章）和柏林噪声（第 14 章）出现的问题进行了非常详细的修正。他们的建议帮助我们对本书进行了改进。我们还收到了许多来自不同学校的学生提出的问题，这些问题帮助我们评估了我们编写方法的优缺点。

我们对本系列书的第一次试用是在 2017 年的秋天，当时我们的同事 Pinar Muyan-Ozcelik 博士在她教授的 CSc-155 课程上第一次使用了 *Computer Graphics Programming in OpenGL with Java*，这让我们有机会评估我们是否实现了让本书成为“自学”资源的目标。课程进展顺利的同时，Pinar Muyan-Ozcelik 博士也为每章留存了问题和更正日志。这份日志帮我们对本书进行了许多改进。

Martín Lucas Golini 是 SOIL2 纹理图像处理库的开发者和维护者，也对本书表现出了极大的支持和热情。我们对他的帮助表示非常感谢。

Jay Turberville 来自于亚利桑那州 Scottsdale 的 Studio 522 Productions。他创建了本书英文版的封面和书中用到的海豚模型，学生们非常喜欢。Studio 522 Productions 制作出极高质量的 3D 动画和视频，以及自定义 3D 建模。我们很感谢 Turberville 慷慨地为本书创建这个精美的模型。

我们还要感谢其他一些艺术家和研究人员。他们非常慷慨地让我们使用他们的模型和纹理。来自 Planet Pixel Emporium 的 James Hastings-Trew 提供了许多行星表面纹理。Paul Bourke 允许我们使用他拥有的精彩的星域。斯坦福大学的 Marc Levoy 博士授权我们使用著名的“斯坦福龙”模型。Paul Baker 的凹凸贴图教程是我们在许多例子中使用的“圆环”模型的基础。我们还要感谢 Mercury Learning 允许我们使用《DirectX 12 3D 游戏开发实战》[①]中的一些纹理。

Danny Kopec 博士向我们介绍了 Mercury Learning 公司，并向它的出版商 David Pallai 引荐了我们。Kopec 博士的《人工智能（第 2 版）》[②]的成功出版让我们考虑与 Mercury 合作，我们与 Kopec 的电话交谈也对我们非常有帮助。Kopec 博士的早逝让我们深感悲痛，也对他没有机会看到本书的成书感到遗憾。

---

① 《DirectX 12 3D 游戏开发实战》已由人民邮电出版社出版（ISBN 978-7-115-47921-1）。——编者注

② 《人工智能（第 2 版）》已由人民邮电出版社出版（ISBN 978-7-115-48843-5）。——编者注

最后，我们要感谢 Mercury Learning 的 David Pallai 和 Jennifer Blaney，他们一直保持着对这个项目的热情并引导我们完成了本书的整个出版流程。

勘误

如果你在阅读本书时发现任何错误，请告诉我们！尽管我们尽了最大努力，但本书肯定还有错误。当收到错误报告时，我们将会尽最大努力尽快发布。我们建立了一个用于收集并发布勘误的网页：

http://athena.ecs.csus.edu/~gordonvs/errata.html[①]

出版商 Mercury Learning 也保留了本书勘误表页面的链接。因此，如果我们的勘误页面的 URL 有变动，请查看 Mercury Learning 网站以获取最新链接。

参考资料

[AS14] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with WebGL*, 7th ed. (Pearson, 2014).

[KS16] J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9th ed. (Addison-Wesley, 2016).

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

[WO13] D. Wolff, *OpenGL Shading Language Cookbook*, 2nd ed. (Packt Publishing, 2013).

① 本书中文版已经对这些勘误进行了修改。——编者注

作者简介

V.斯科特·戈登（V. Scott Gordon）博士已经在加州州立大学系统担任教授有 20 多年，目前在加州州立大学萨克拉门托分校教授高级图形和游戏工程课程。他撰写及合著了 30 多部出版物，涉及人工智能、神经网络、进化计算、软件工程、视频和策略游戏编程，以及计算机科学教育等多个领域。戈登博士在科罗拉多州立大学获得博士学位。他同时也是爵士鼓手和优秀的乒乓球运动员。

约翰·克莱维吉（John Clevenger）博士拥有超过 40 年的教学经验，教学内容包括高级图形、游戏架构、操作系统、VLSI 芯片设计、系统仿真和其他主题。他是多个用于图形和游戏架构教学的软件框架和工具的开发人员，其中包括我们 Java 版第一版书中所用到的 graphicslib3D 库。他是国际大学生程序设计竞赛（ICPC）的技术总监，负责监督 PC^2 的持续开发。PC^2 是目前世界上使用较为广泛的编程竞赛支持系统。克莱维吉博士在加州大学戴维斯分校获得博士学位。

资源与支持

本书由异步社区出品，社区（https://www.epubit.com/）为您提供相关资源和后续服务。

配套资源

本书提供如下资源：

- 本书配套源代码；
- 书中用到的模型、图形、纹理和贴图等；
- 书中彩图文件。

要获得以上配套资源，请在异步社区本书页面中单击 配套资源 ，跳转到下载界面，按提示进行操作即可。注意：为保证购书读者的权益，该操作会给出相关提示，要求输入提取码进行验证。

如果您是教师，希望获得教学配套资源，请在社区本书页面中直接联系本书的责任编辑。

提交勘误

作者和编辑尽最大努力来确保书中内容的准确性，但难免会存在疏漏。欢迎您将发现的问题反馈给我们，帮助我们提升图书的质量。

当您发现错误时，请登录异步社区，按书名搜索，进入本书页面，单击“提交勘误”，输入勘误信息，单击“提交”按钮即可。本书的作者和编辑会对您提交的勘误进行审核，确认并接受后，您将获赠异步社区的 100 积分。积分可用于在异步社区兑换优惠券、样书或奖品。

详细信息　写书评　提交勘误

页码：　页内位置（行数）：　勘误印次：

字数统计

提交

扫码关注本书

扫描下方二维码，您将会在异步社区微信服务号中看到本书信息及相关的服务提示。



与我们联系

我们的联系邮箱是 contact@epubit.com.cn。

如果您对本书有任何疑问或建议，请您发邮件给我们，并请在邮件标题中注明本书书名，以便我们更高效地做出反馈。

如果您有兴趣出版图书、录制教学视频，或者参与图书翻译、技术审校等工作，可以发邮件给我们；有意出版图书的作者也可以到异步社区在线提交投稿（直接访问 www.epubit.com/selfpublish/submission 即可）。

如果您是学校、培训机构或企业，想批量购买本书或异步社区出版的其他图书，也可以发邮件给我们。

如果您在网上发现有针对异步社区出品图书的各种形式的盗版行为，包括对图书全部或部分内容的非授权传播，请您将怀疑有侵权行为的链接发邮件给我们。您的这一举动是对作者权益的保护，也是我们持续为您提供有价值的内容的动力之源。

关于异步社区和异步图书

“异步社区”是人民邮电出版社旗下 IT 专业图书社区，致力于出版精品 IT 技术图书和相关学习产品，为作译者提供优质出版服务。异步社区创办于 2015 年 8 月，提供大量精品 IT 技术图书和电子书，以及高品质技术文章和视频课程。更多详情请访问异步社区官网 https://www.epubit.com。

“异步图书”是由异步社区编辑团队策划出版的精品 IT 专业图书的品牌，依托于人民邮电出版社近 30 年的计算机图书出版积累和专业编辑团队，相关图书在封面上印有异步图书的 LOGO。异步图书的出版领域包括软件开发、大数据、AI、测试、前端、网络技术等。



异步社区



微信服务号

目　录

第 1 章　入门

图形编程是计算机科学中最具挑战性的主题之一，并因此而闻名。当今，图形编程是基于着色器的——也就是说，有些程序是用诸如 C++或 Java 等标准编程语言编写的，并运行在 CPU 上；而另一些是用专用的着色器语言编写的，并直接运行在显卡（GPU）上。着色器编程的学习曲线很陡峭，以致哪怕是绘制简单的东西，也需要一系列错综复杂的步骤，把图形数据从一个“管线”中传递下去才能完成。现代显卡能够并行处理数据，即使是绘制简单的形状，图形程序员也必须理解 GPU 的并行架构。

虽然这并不简单，但回报是超强的渲染能力。电子游戏中涌现出来的令人惊艳的虚拟现实和好莱坞电影中越来越逼真的特效，很大程度上是由着色器编程的进步带来的。如果阅读本书是你进入 3D 图形世界的第一步，那么你正在开始接受一个对自己的挑战。挑战的奖励不仅仅是漂亮的图片，还有过往不敢想象的对机器的掌控程度。欢迎来到激动人心的计算机图形编程世界！

1.1　语言和库

现代图形编程使用图形库完成，也就是说，程序员编写代码时，调用一个预先定义的库（或者一系列库）中的函数，由这个库来提供对底层图形操作的支持。现在有很多图形库，但常见的平台无关图形编程库叫作 OpenGL（Open Graphics Library，开放图形库）。本书将会介绍如何在 C++中使用 OpenGL 进行 3D 图形编程。

在 C++中使用 OpenGL 需要配置多个库。这里按照个人需求，可以有一系列令人眼花缭乱的选择。在本节中，我们会介绍哪几种库是必要的，各种库的一些常见选择，以及我们在本书中选择的库。

总的来说，你需要以下这些语言和库：

- C++开发环境；
- OpenGL / GLSL；
- 窗口管理；
- 扩展库；
- 数学库；
- 纹理管理。

读者可能需要进行几个准备步骤，以保证这几种库已安装在系统中，并可以正常使用。下面几个小节将简单介绍每一种语言和库。安装和配置的更多细节请参阅附录。

1.1.1 C++

C++是一种通用编程语言，最早出现在20世纪80年代中期。它的设计，以及它通常被编译成本机的机器码这一事实，使得它成为了需要高性能的系统的优秀选择，比如3D图形计算。C++的另一个优点是OpenGL调用库是基于C语言开发的。

有许多可用的C++开发环境。在阅读本书时，如果读者使用PC（Windows操作系统），我们推荐使用Microsoft Visual Studio [VS17]；如果在苹果计算机上，我们推荐Xcode [XC18]。附录中也介绍了各个平台下的安装和配置。

1.1.2 OpenGL / GLSL

OpenGL的1.0版本出现在1992年，是一种对供应商特定的计算机图形应用编程接口（API）的“开放性”替代。

它的规范和开发工作由当时新成立的OpenGL架构评审委员会（ARB）管理和控制。ARB是一群行业参与者组成的小组。2006年，ARB将OpenGL规范的控制权交给了Khronos Group。Khronos Group是一个非营利性联盟，不仅管理OpenGL标准，还管理很多其他的开放性行业标准。

从一开始，OpenGL就定期修订和扩展。2004年，2.0版本中引入了OpenGL着色语言（GLSL），使得“着色器程序”可以在图形管线的各个阶段被安装和直接执行。

2009年，3.1版本中移除了大量被弃用的功能，以强制使用着色器编程，而不是之前的老方法（叫作“立即模式”）。①在最近的功能中，4.0版本（2010年）在可编程管线中增加了一个细分阶段。

这本书假定用户的机器有一个支持至少4.3版本OpenGL的显卡。如果你不确定你的GPU支持哪个版本的OpenGL，网上有免费的应用程序可以用来找出答案。有一个这样的应用程序是GLView，由“realtech VR”公司提供[GV16]。

1.1.3 窗口管理

OpenGL实际上并不是把图像直接绘制到计算机屏幕上，而是渲染到一个帧缓冲区，然后需要由这台机器来负责把帧缓冲区的内容绘制到屏幕上的一个窗口中。有不少库都可以支持这一部分工作。一个选择是使用操作系统提供的窗口管理功能，比如**Microsoft Windows API**。但这通常是不实用的，需要很多底层的编码工作。GLUT库曾经是一个很流行的选择，但现在已经被弃用了。它的一个现代化的演变是freeglut库。其他相关的选项还有CPW库、GLOW库和GLUI库。

GLFW是最流行的选择之一，也是我们这本书中选择使用的。它内置了对Windows、macOS、Linux和其他操作系统[GF17]的支持。它可以在其官网下载，并且必须在要使用它的机器上编译。（我们在附录中介绍了相关步骤。）

① 尽管如此，许多显卡厂商（比如NVIDIA）依然继续支持被弃用的功能。

1.1.4 扩展库

OpenGL 围绕一组基本功能和扩展机制进行组织。随着技术的发展，扩展机制可以用来支持新的功能。现代版本的 OpenGL，比如我们在本书中使用的 4 以上版本，需要识别 GPU 上可用的扩展。OpenGL 核心中有一些内置的命令用来支持这些，但是为了使用每个现代命令，需要执行很多相当复杂的代码行。在本书中，我们会持续不断地使用这些命令。所以使用一个*扩展库*来处理这些细节已经成了标准做法，这样能让程序员可以直接使用现代 OpenGL 命令。比如 Glee、GLLoader 和 GLEW，以及更加新的 GL3W 和 GLAD。

列出的这些库中，常用的是 GLEW，意思是 OpenGL *扩展牧马人*（OpenGL Extension Wrangler）。它支持各种操作系统，包括 Windows、Macintosh 和 Linux [GE17]。GLEW 不是一个完美的选择。例如，它需要一个额外的 DLL。最近，很多开发者选择 GL3W 或者 GLAD。它们的优势是可以自动更新，但是要求安装 Python。因为这些原因，在本书中我们选择使用 GLEW。它可以在官网下载。附录中给出了安装和配置 GLEW 的完整说明。

1.1.5 数学库

3D 图形编程大量使用向量和矩阵代数。因此，配合一个支持常见数学计算任务的函数库或者类包，能极大地方便 OpenGL 的使用。常常和 OpenGL 一起使用的两个这样的库是 Eigen 和 vmath。后者在流行的 OpenGL SuperBible [SW15]中被使用。

可能最流行的数学库，也是本书中使用的，是 OpenGL Mathematics，一般称作 GLM。它是一个只有头文件的 C++库，兼容 Windows、macOS 和 Linux [GM17]。GLM 命令很方便地遵循和 GLSL 相同的命名惯例，使得来回阅读特定应用程序的 C++和 GLSL 代码时更容易。GLM 可以在官网下载。

GLM 提供与图形概念相关的类和基本数学函数，例如*矢量*、*矩阵*和*四元数*。它还包含各种工具类，用于创建和使用常见的 3D 图形结构，例如透视和视角矩阵。它最早在 2005 年发布，由 Christophe Riccio [GM17]维护。有关安装 GLM 的说明，请参阅附录。

1.1.6 纹理管理

从第 5 章开始，我们将使用图像文件来向我们图形场景中的对象添加“纹理”。这意味着我们会需要频繁加载这些图像文件到我们的 C++ / OpenGL 代码中。从零开始写一个纹理图像加载器是可能的。但是，考虑到各种各样的图像文件格式，使用一个纹理加载库通常是更好的。比如 FreeImage、DevIL、OpenGL Image (GLI)和 Glraw。简单 OpenGL 图像加载器（Simple OpenGL Image Loader，SOIL）可能是最常用的 OpenGL 图像加载库，尽管它有点过时了。

本书中使用的纹理图像加载库是 SOIL2——SOIL 的一个更新的分叉版本。像我们之前选择的库一样，SOIL2 兼容各种平台[SO17]。附录中给出了详细的安装和配置说明。

1.1.7 可选库

读者可能希望利用很多其他有用的库。例如，在本书中，我们将展示如何从零开始实现一个简单的“OBJ”模型加载器。然而，正如我们将看到的，它没有处理 OBJ 标准中可用的很多选项。有一些更复杂的现成的 OBJ 加载器可供选择，比如 Assimp 和 tinyobjloader。在我们的例子中，我们会只用在本书中介绍和实现的简单模型加载器。

1.2 安装和配置

在开发本书的 C++版本时，我们斗争了很久，想要找到囊括用来运行示例程序的平台特定配置信息的最佳方法。配置用 C++来使用 OpenGL 的系统，要比用 Java 配置复杂得多。Java 版本的配置只需要几个短段落就可以描述完毕（正如在本书 Java 版中看到的[GC18]）。最终，我们选择把安装和配置信息在各平台特定的附录中分别描述。我们希望这能为每个读者提供一个相关的地方来寻找关于他/她的系统的特定信息，而不是被和他/她无关的其他平台的信息干扰。在这个版本中，我们在附录 A 中提供了 Microsoft Windows 平台的详细配置教程，在附录 B 中提供了苹果 Macintosh 平台的详细配置教程。

参考资料

[GC18] V. Gordon and J. Clevenger, *Computer Graphics Programming in OpenGL with Java*, 2nd ed. (Mercury Learning, 2018).

[GE17] OpenGL Extension Wrangler (GLEW), accessed October 2018.

[GF17] Graphics Library Framework (GLFW), accessed October 2018.

[GM17] OpenGL Mathematics (GLM), accessed December 2017.

[GV16] GLView, realtech-vr, accessed October 2018.

[SO17] Simple OpenGL Image Library 2 (SOIL2), *SpartanJ*, accessed October 2018.

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

[VS17] Microsoft Visual Studio downloads, accessed October 2018.

[XC18] Apple Developer site for Xcode, accessed January 2018.

第2章 OpenGL 图像管线

OpenGL 是整合软硬件的多平台 2D 和 3D 图形 API。使用 OpenGL 需要显卡（GPU）支持足够新版的 OpenGL（如第 1 章所述）。

在硬件方面，OpenGL 提供了一个多级图形管线，可以使用一种名为 GLSL 的语言进行部分编程。

软件方面，OpenGL 的 API 是用 C 语言编写的，因此 API 调用直接兼容 C 和 C++。对于十几种其他的流行语言（Java、Perl、Python、Visual Basic、Delphi、Haskell、Lisp、Ruby 等），OpenGL 也有着稳定的库（或“包装器”），具有与 C 语言库几乎相同的性能。本书使用的 C++，应该是目前流行的 OpenGL 语言。使用 C++时，程序员编写在 CPU 上运行的（编译后的）代码并包含 OpenGL 调用。当一个 C++程序包含 OpenGL 调用时，我们将其称为 C++/OpenGL 应用程序。C++/OpenGL 应用程序的一个重要任务是将程序员的 GLSL 代码安装到 GPU 上。

基于 C++的图形应用大致如图 2.1 所示，其中软件部分以底色突出显示。

在我们后面的代码中，一部分用 C++编码，进行 OpenGL 调用；另一部分是 GLSL。C++/OpenGL 应用程序、GLSL 模块和硬件一起用来生成 3D 图形输出。当应用完成之后，最终用户直接与 C++应用程序进行交互。

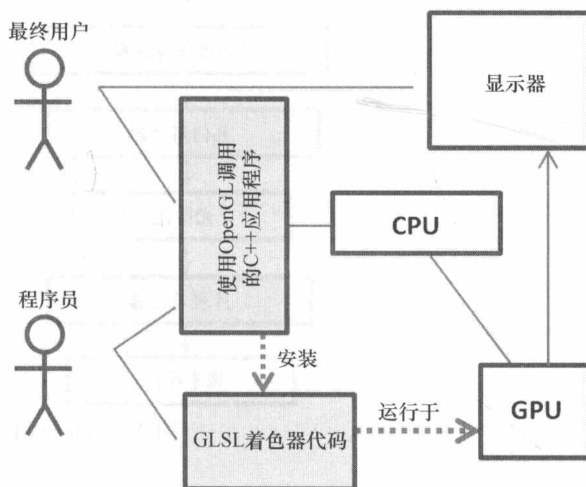


图 2.1 基于 C++的图形应用概览

GLSL 是一种着色器语言。着色器语言主要运行于 GPU 上，在图形管线上下文中。还有一些其他的着色器语言，如 HLSL，用于微软的 3D 框架 DirectX。GLSL 是与 OpenGL 兼容的专用着色器语言，因此我们在 C++/OpenGL 应用代码之外，需要用 GLSL 写着色器代码。

本章其余内容将简单地浏览 OpenGL 管线的内容。读者不用期望详细理解所有细节，这里只要对各阶段如何工作有大致感觉即可。

2.1 OpenGL 管线

现代 3D 图形编程会使用管线的概念，在管线中。将 3D 场景转换成 2D 图形的过程被分割成许多步骤。OpenGL 和 DirectX 使用了相似的管线概念。

图 2.2 展示了 OpenGL 图形管线简化后的概览（并未展示所有阶段，仅包含我们要学习的主要阶段）。C++/OpenGL 应用发送图形数据到顶点着色器，随着管线处理，最终生成在显示器上显示的像素点。

用灰色阴影表示的阶段（顶点着色器、曲面细分着色器、几何着色器、片段着色器）可以用 GLSL 进行编程。将 GLSL 程序载入这些着色器阶段也是 C++/OpenGL 程序的责任之一，其过程如下。

（1）首先使用 C++获取 GLSL 着色器代码，既可以从文件中读取，也可以硬编码在字符串中。

（2）接下来创建 OpenGL 着色器对象并将 GLSL 着色器代码加载进着色器对象。

（3）最后，用 OpenGL 命令编译并连接着色器对象，并将它们安装进 GPU。

在实践中，一般至少要提供顶点着色器和片段着色器阶段的 GLSL 代码，而曲面细分着色器和几何着色器阶段是可选的。接下来我们将简单地过一下整个过程，并看看每步发生了什么。

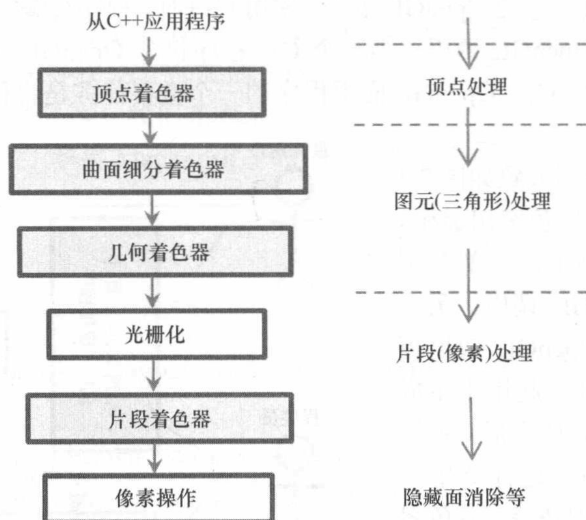


图 2.2 OpenGL 管线概览

2.1.1 C++/OpenGL 应用程序

我们的图形应用程序中大部分是使用 C++进行编写的。根据程序目的的不同，它可能需要用标准 C++库与最终用户交互，用 OpenGL 调用实现与 3D 渲染相关的任务。正如前面章节所述，我们将会使用一些扩展库：GLEW（OpenGL Extension Wrangler）、GLM（OpenGL Mathmatics）、SOIL2（Simple OpenGL Image Loader）以及 GLFW（Graphics Library Framework）。

GLFW 库包含了 GLFWwindow 类，我们可以在其上进行 3D 场景绘制。如前所述，OpenGL 也向我们提供了用于将 GLSL 程序安装到可编程着色器阶段并编译的命令。最后，OpenGL 使用缓冲区将 3D 模型和其他相关图形数据发送到管线中。

在我们尝试编写着色器之前，先写一个简单的 C++/OpenGL 程序，创建一个 GLFWwindow 实例并为其设置背景色。这个过程根本用不到着色器！其代码如程序 2.1 所

示。程序 2.1 中的 main()函数与本书中所有将会用到的 main()函数一样。其中重要的操作有：（a）初始化 GLFW 库；（b）实例化 GLFWwindow；（c）初始化 GLEW 库；（d）调用一次 init()函数；（e）重复调用 display()函数。

我们将每个应用程序的初始化任务都放在 init()函数中，用于绘制 GLFWwindow 的代码都将放在 display()函数中。

在本例中，glClearColor()命令指定了清除背景时用的颜色值——这里(1,0,0,1)代表 RGB 值中的红色（末尾的 1 表示不透明度）。接下来使用 OpenGL 调用 glClear(GL_COLOR_BUFFER_BIT)，实际使用红色对颜色缓冲区进行填充。

程序 2.1 第一个 C++/OpenGL 应用程序

```
#include <GL\glew.h>
#include <GLFW\glfw3.h>
#include <iostream>

using namespace std;

void init(GLFWwindow* window) { }

void display(GLFWwindow* window, double currentTime) {
   glClearColor(1.0, 0.0, 0.0, 1.0);
   glClear(GL_COLOR_BUFFER_BIT);
}

int main(void) {
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 - program1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {
        display(window, glfwGetTime());
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

图 2.3 展示了程序 2.1 的输出。

这些函数部署的机制如下：GLFW 和 GLEW 库先分别使用 glfwInit()和 glewInit()初始化。glfwCreateWindow()命令负责创建 GLFW 窗口，同时其相关的 OpenGL 上下文[①]由 glfwCreateWindow()命令创建，其可选项由前面的 WindowHints 设置。WindowHints 指定了机器必须与 OpenGL 版本 4.3 兼容（“主版本号”=4，“次版本号”=3）。glfwCreateWindow

① “上下文”是指 OpenGL 实例及其状态信息，其中包括诸如颜色缓冲区之类的项。

命令的参数指定了窗口的宽、高（以像素为单位）以及窗口顶部的标题。（这里没有用到的另外两个参数设为 NULL，分别用来允许全屏显示以及资源共享。）glfwSwapInterval()命令和 glfwSwapBuffers 命令用来开启垂直同步（Vsync）——GLFW 窗口默认是双缓冲的。[①]这里需要注意，创建 GLFW 窗口并不会自动将它与当前 OpenGL 上下文关联起来——因此我们需要调用 glfwMakeContextCurrent()。

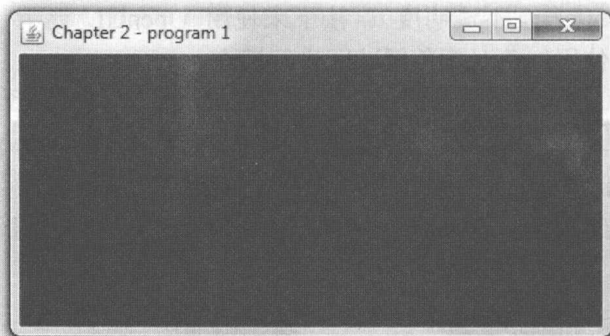


图 2.3 程序 2.1 的输出

main()函数包括了一个简单的渲染循环，用来反复调用 display()方法。同时它也调用了 glfwSwapBuffers()以绘制屏幕，以及 glfwPollEvents()以处理窗口相关事件（如按键事件）。当 GLFW 探测到应该关闭窗口的事件（如用户单击了右上角的×）时，循环就会终止。这里需要注意，我们将一份 GLFW 窗口对象的引用传入了 init()和 display()调用。这些函数在特定环境下需要访问 GLFW 窗口对象。同时我们也将当前时间传入了 display()调用，这样方便保证动画在不同计算机上以相同速度播放。在这里，我们用了 glfwGetTime()，它会返回 GLFW 初始化之后经过的时间。

现在是时候详细看看程序 2.1 中的 OpenGL 调用了。首先关注一下这个调用：

```
glClear(GL_COLOR_BUFFER_BIT);
```

在这里，调用的 OpenGL 参考文档中的描述是：

```
void glClear(GLbitfield mask);
```

参数中引用了类型为 GLbitfield 的“GL_COLOR_BUFFER_BIT”。OpenGL 有很多预定义的常量（其中很多是枚举量）。GL_COLOR_BUFFER_BIT 引用了包含渲染后像素的颜色缓冲区。OpenGL 有多个颜色缓冲区，这个命令会将它们全部清除——用一种被称为“清除色（clear color）”的预定义颜色填充所有缓冲区。注意，这里的“清除（clear）”表示的不是“颜色清晰”，而是用来重置缓冲区时填充的颜色（清除）。

在调用 glClear()后紧接着是 glClearColor()的调用。glClearColor()让我们能够指定颜色缓冲区清除后填充的值。这里我们指定了(1,0,0,1)，即 RGBA 颜色中的红色。

---

① “双缓冲”意味着有两个颜色缓冲区——一个显示，一个渲染。渲染整个帧后，将交换缓冲区。缓冲用于减少不良的视觉伪影。

最后，当用户尝试关闭 GLFW 窗口时，程序将退出渲染循环。这时，main()会通过分别调用 glfwDestroyWindow()和 glfwTerminate()通知 GLFW 销毁窗口以及终止运行。

2.1.2 顶点着色器和片段着色器

在第一个 OpenGL 程序中，我们实际上并没有绘制任何东西——仅仅用一种颜色来填充了颜色缓冲区。要真的绘制点什么，我们需要加入顶点着色器和片段着色器。

你可能对于学习 OpenGL 只让你绘制少数几类非常简单的东西有点吃惊，如点、线、三角形。这些简单的东西叫作图元，多数 3D 模型通常是由许多三角形的图元构成。图元由顶点组成——例如三角形有 3 个顶点。顶点可以由很多来源产生——从文件读取并由 C++/OpenGL 应用载入缓冲区、直接在 C++文件中硬编码或者直接在 GLSL 代码中。

在加载顶点之前，C++/OpenGL 应用必须编译并链接合适的 GLSL 顶点着色器和片段着色器程序，之后将它们载入管线。我们稍后将会看到这些命令。

C++/OpenGL 应用同时也负责通知 OpenGL 构建三角形，通过使用如下 OpenGL 函数完成：

```
glDrawArrays(GLenum mode, Glint first, GLsizei count);
```

mode 参数是图元的类型——对于三角形我们用 GL_TRIANGLES。first 参数表示从哪个顶点开始绘制（通常是顶点 0，即第一个顶点），count 表示总共要绘制的顶点数。

当调用 glDrawArrays()时，管线中的 GLSL 代码开始执行。现在可以向管线加一些 GLSL 代码了。

不管它们从何处读入，所有的顶点都会被传入顶点着色器。顶点们会被一个一个地处理，即着色器会对每个顶点执行一次。对拥有很多顶点的大型复杂模型而言，顶点着色器会执行成百上千甚至百万次，这些执行过程通常是并行的。

现在，我们来写一个简单的程序，它仅包含一个顶点，硬编码于顶点着色器中。虽然这不足以让我们画三角形，但是足够画出一个点。为了显示这个点，我们还需要提供片段着色器。为简单起见，我们将这两个着色器程序声明为字符串数组。

程序 2.2 着色器，画一个点

```
(#include 列表与之前相同)
#define numVAOs 1
                                  } 新的定义
GLuint renderingProgram;
GLuint vao[numVAOs];

GLuint createShaderProgram() {
  const char *vshaderSource =
    "#version 430 \n"
    "void main(void) \n"
    "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";

  const char *fshaderSource =
    "#version 430 \n"
    "out vec4 color; \n"
    "void main(void) \n"
    "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";
```

```
  GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
  GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);

  glShaderSource(vShader, 1, &vshaderSource, NULL);
  glShaderSource(fShader, 1, &fshaderSource, NULL);
  glCompileShader(vShader);
  glCompileShader(fShader);

  GLuint vfProgram = glCreateProgram();
  glAttachShader(vfProgram, vShader);
  glAttachShader(vfProgram, fShader);
  glLinkProgram(vfProgram);

  return vfProgram;
}

void init(GLFWwindow* window) {
  renderingProgram = createShaderProgram();
  glGenVertexArrays(numVAOs, vao);
  glBindVertexArray(vao[0]);
}

void display(GLFWwindow* window, double currentTime) {
  glUseProgram(renderingProgram);
  glDrawArrays(GL_POINTS, 0, 1);
}

...main()函数与之前相同
```

程序看起来只显示了一个空的窗口（见图 2.4）。但仔细观察一下，会发现窗口中央有一个蓝色的点（假设本页印刷精度足够）。OpenGL 中点的默认大小为 1 像素。

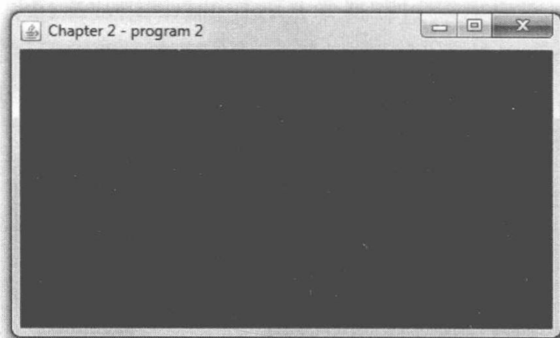


图 2.4 程序 2.2 的输出效果

程序 2.2 中有很多值得讨论的重要细节（为方便起见已用颜色标出）。首先，注意其中多次用到的“Gluint”——这是由 OpenGL 提供的“unsigned int”的平台无关简写（许多 OpenGL 结构体都是整数类型引用）。接下来，init()不再是空函数了——现在它会调用另一个叫作“createShaderProgram”的函数（我们写的）。“createShaderProgram”函数先定义了两个字符串 vshaderSource 和 fshaderSource。之后调用了两次 glCreateShader()函数，创建了类型为 GL_VERTEX_ SHADER 和 GL_FRAGMENT_SHADER 的两个着色器。OpenGL 创建每个着色器对象（初始值为空）的时候，会返回一个整数 ID 作为后面引用它的序号——我们的代

码将这个 ID 存入了 vShader 和 fShader 变量中。之后，createShaderProgram()调用了 glShaderSource()，这个函数用于将 GLSL 代码从字符串载入空着色器对象中。之后，用 glCompileShader()编译各着色器。glShaderSource()有 4 个参数：（a）用来存放着色器的着色器对象，（b）着色器源代码中的字符串数量，（c）包含源代码的字符串指针，（d）最后一个没用到的参数（我们稍后会在补充章节说明中解释这个参数）。注意，这两次调用 glCompileShader()时都指明了着色器的源代码字符串数量为“1”——这个参数也会在补充说明中解释。

之后应用程序创建了一个叫作 vfProgram 的程序对象，并储存指向它的整数 ID。OpenGL“程序”对象包含一系列编译过的着色器，这里可以看到使用 glCreateProgram()创建程序对象，使用 glAttachShader()将着色器加入程序对象，之后使用 glLinkProgram()来请求 GLSL 编译器确保它们的兼容性。

如前所见，在 init()结束后程序调用了 display()。display()函数所做的事情中包含调 glUseProgram()，它将含有两个已编译着色器的程序载入 OpenGL 管线阶段（在 GPU 上！）。注意 glUseProgram()并没有运行着色器，它只是将着色器加载进硬件。

我们稍后在第 4 章会看到，一般情况下，这里 C++/OpenGL 将会准备要发送给管线绘制的模型的顶点集。但是本例中，由于是第一个着色器程序，我们仅仅在顶点着色器中硬编码了一个顶点。因此，本例中的 display()函数接着调用了 glDrawArrays()用来启动管线处理过程。原始类型是 GL_POINTS，仅用来显示一个点。

现在我们来看一下着色器，在之前用绿色展示（并在之后的解释中又重复了一遍）。正如我们所看到的，在 C++/OpenGL 程序中，它们声明为字符串数组。这是一种笨拙的编程方式，不过在这个超简单的例子中足够了。这个顶点着色器是：

```
#version 430
void main(void)
{    gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }
```

第一行指明了 OpenGL 版本，这里是 4.3。接下来是一个“main”函数（我们后面将会看到，GLSL 句法上与 C++类似）。所有顶点着色器的主要目标都是将顶点发送给管线（正如之前所说的，它会对每个顶点进行处理）。内置变量 gl_Position 用来设置顶点在 3D 空间的坐标位置，并发送至下一个管线阶段。GLSL 数据类型 vec4 用来存储 4 元组，适合用来存储坐标，4 元组的前 3 个值分别表示 *X*、*Y*、*Z*，第 4 个值在这里设为 1.0（第 3 章将会学习第 4 个值的用途）。本例中，顶点坐标硬编码于原点(0,0,0)。

顶点接下来将沿着管线移动到光栅着色器，它们会在这里被转换成像素位置（更精确地说是片段——后面会解释）。最终这些像素（片段）到达片段着色器：

```
#version 430
out vec4 color;
void main(void)
{    color = vec4(0.0, 0.0, 1.0, 1.0); }
```

所有片段着色器的目的都是给将要展示的像素赋予 RGB 颜色。在本例中所指定的输出颜色值(0,0,1)是蓝色（第 4 个值 1.0 是不透明度）。注意这里的“out”标签表明 color 变量是输出变量。（在顶点着色器中并不是必须给 gl_Position 指定“out”标签，因为 gl_Position

是预定义的输出变量。)

代码中还有一处我们没有讨论的细节，即 init()函数中的最后两行（以红色显示）。它们看起来可能有些神秘。我们在第 4 章中将会看到，当准备将数据集发送给管线时是以缓冲区形式发送的。这些缓冲区最后都会被存入顶点数组对象（Vertex Array Object，VAO）中。在本例中，我们向顶点着色器中硬编码了一个点，因此我们不需要任何缓冲区。但是，即使应用程序完全没有用到任何缓冲区，OpenGL 仍然需要在使用着色器的时候至少有一个创建好的 VAO，所以这两行用来创建 OpenGL 要求的 VAO。

最后的问题就是从顶点着色器出来的顶点是如何变成片段着色器中的像素的。回忆一下图 2.2 中，在顶点处理和像素处理中间存在着光栅化阶段。正是在这个阶段中图元（如点或三角形）转换成了像素集合。OpenGL 中默认点的大小为 1 像素，这就是为什么我们的单点最终渲染成了单个像素。

让我们将下面的命令加入 display()函数中，就放在调用 glDrawArrays()之前：

```
glPointSize(30.0f);
```

现在，当光栅化阶段从顶点着色器收到顶点时，它会为一个大小是 30 像素的点设置像素颜色值。输出的结果展示在图 2.5 中（见彩插）。



图 2.5 改变 glPointSize

让我们继续观察剩下的 OpenGL 管线。

2.1.3 曲面细分着色器

我们在第 12 章中介绍曲面细分。可编程曲面细分阶段是最近加入 OpenGL（在 4.0 版中）的功能。它提供了一个*曲面细分着色器*用以生成大量三角形，通常是网格形式。同时也提供一些可以以各种方式操作这些三角形的工具。例如，程序员可能需要以图 2.6 展示的方式操作一个曲面细分过的三角形网格。

当在简单形状上需要很多顶点时，曲面细分着色器就能发挥作用了，如在方形区域或曲面上。稍后我们会看到，它在生成复杂地形时也很有用。对于这种情况，有时用 GPU 中的曲面细分着色器在硬件里生成三角形网格比在 C++中生成要高效得多。

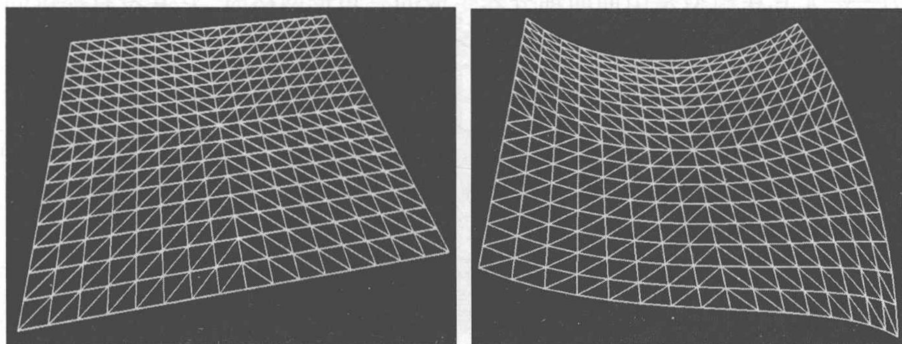


图 2.6　曲面细分着色器生成的网格

2.1.4　几何着色器

我们在第 13 章中介绍了几何着色器阶段。顶点着色器赋予程序员一次操作一个顶点的能力（“按顶点”处理），片段着色器（稍后会看到）允许一次操作一个像素（“按片段”处理），几何着色器赋予了一次操作一个图元的能力（“按图元”处理）。

回顾前文提到最通用的图元是三角形。当我们到达几何着色器阶段时，管线肯定已经完成了将顶点组合为三角形的过程（这个过程叫作图元组装）。接下来几何着色器会让程序员可以同时访问每个三角形的所有顶点。

按图元处理有很多用途，既可以让图元变形，比如拉伸或者缩小，还可以删除一些图元，从而在渲染的物体上产生“洞”——这是一种将简单模型转化为复杂模型的方法。

几何着色器也提供了生成额外图元的方法。这些方法也打开了通过转换简单模型而得到复杂模型的大门。几何着色器有一种有趣的用法，就是在物体上增加表面纹理，如凸起、鳞甚至“毛发”。考虑图 2.7 所示的简单环面（本书后面会介绍如何生成它）。环面的表面由上百个三角形构成。如果我们用几何着色器对每个三角形外面增加一个额外的三角形，就会得到如图 2.8 所示的结果。这个“鳞环面”如果是从 C++/OpenGL 应用程序那边从零建模生成，代价就大了。

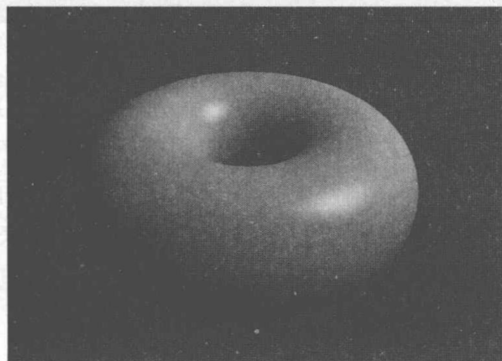


图 2.7　环面模型

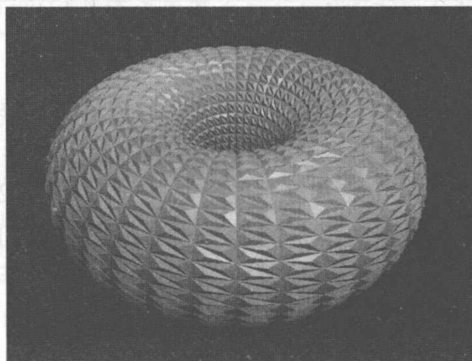


图 2.8　几何着色器修改后的环面

在曲面细分阶段已经给程序员同时访问模型中所有顶点的能力后，再提供一个按图元运算的着色器阶段可能看起来有点多余。它们的区别是，曲面细分只在非常少数情况下提供

了这个能力——尤其在模型是由曲面细分器生成的三角形网格时。它并没有提供同时访问所有顶点，即任何从C++用缓冲区传来的顶点的能力。

2.1.5 光栅化

最终，我们3D世界中的点、三角形、颜色等全都需要展现在一个2D显示器上。这个2D屏幕由光栅——矩形像素阵列组成。

当3D物体光栅化后，OpenGL将物体中的图元（通常是三角形）转化为片段。片段拥有关于像素的信息。光栅化过程确定了用以显示3个顶点所确定的三角形的所有像素需要绘制的位置。

光栅化过程开始时先对三角形的每对顶点进行插值。插值过程可以通过选项调节。就目前而言，使用图2.9所示的简单的线性插值就够了。原本的3个顶点标记为红色（见彩插）。

如果光栅化过程到此为止，那么呈现出的图像将会是线框模型。呈现线框模型也是OpenGL中的一个选项。通过在display()函数中glDrawArrays()调用之前添加如下命令：

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

如果2.1.4小节中的环面使用了这行额外代码，它将会看起来如图2.10所示。

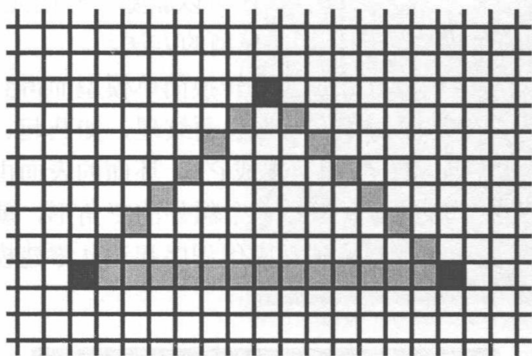


图2.9 光栅化（步骤1）

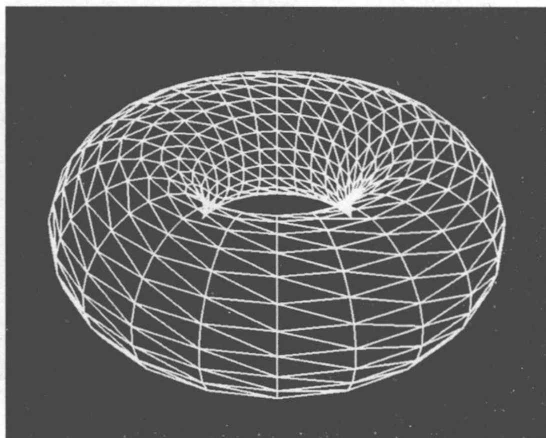


图2.10 使用线框模型渲染的环面

如果我们不加入之前的那一行代码（或者我们在其中使用GL_FILL而非GL_LINE），插值过程将会继续沿着光栅线填充三角形的内部，如图2.11所示。当应用于环面时会产生一个完全光栅化的“实心”环面，如图2.12（左）所示。请注意，在这种情况下，环面的整体形状和曲率不明显——这是因为我们没有包括任何纹理或照明技术，因此它看起来是“平”的。图2.12（右）是同样的“平”环面叠加了线框模型。前面图2.7所示的环面包括了照明效果，因此更清晰地显示了环面的形状。我们将在第7章学习照明。

在本章后面我们将看到，光栅化不仅可以对像素插值。任何顶点着色器输出的变量和片段着色器的输入变量都可以基于对应的像素进行插值。我们将会使用该功能生成平滑的颜色渐变，实现真实光照以及许多其他效果。

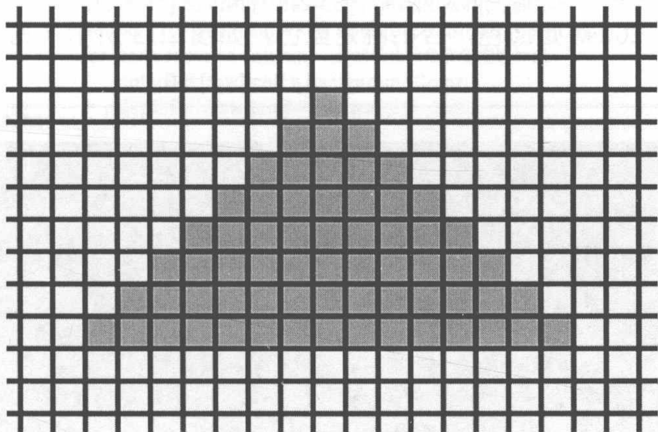


图 2.11　完全光栅化的三角形

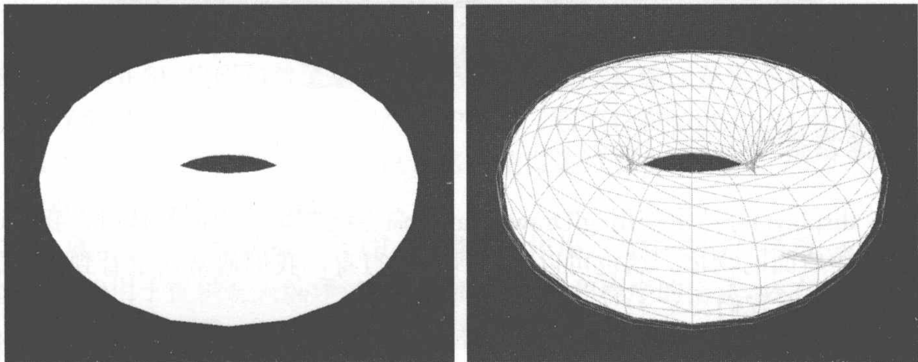


图 2.12　环面的完全光栅化图元渲染（左）和使用线框叠加（右）

2.1.6　片段着色器

如前所述，片段着色器用于为光栅化的像素指定颜色。我们已经在程序 2.2 中看到了片段着色器示例。在程序 2.2 中，片段着色器仅将输出硬编码为特定值，从而为每个输出的像素赋予相同的颜色。不过 GLSL 为我们提供了其他计算颜色的方式，用以表现无穷的创造力。

一个简单的例子就是基于像素位置决定输出颜色。回忆我们在顶点着色器中，顶点的输出坐标使用了预定义变量 gl_Position。在片段着色器中，同样有一个变量让程序员可以访问输入片段的坐标，叫作 gl_FragCoord。我们可以通过修改程序 2.2 中的片段着色器，让它使用 gl_FragCoord（在本例中通过 GLSL 属性选择语法引用它的 *X* 坐标）基于位置设置每个像素的颜色，如：

```
#version 430
out vec4 color;
void main(void)
{ if (gl_FragCoord.x < 200) color = vec4(1.0, 0.0, 0.0, 1.0); else color = vec4(0.0, 0.0, 1.0, 1.0);
}
```

如果我们像在 2.1.2 小节末尾那样增大 GL_PointSize，渲染的点的像素颜色将会以坐标变化——*X* 坐标小于 200 时是红色，否则就是蓝色，如图 2.13 所示（见彩插）。

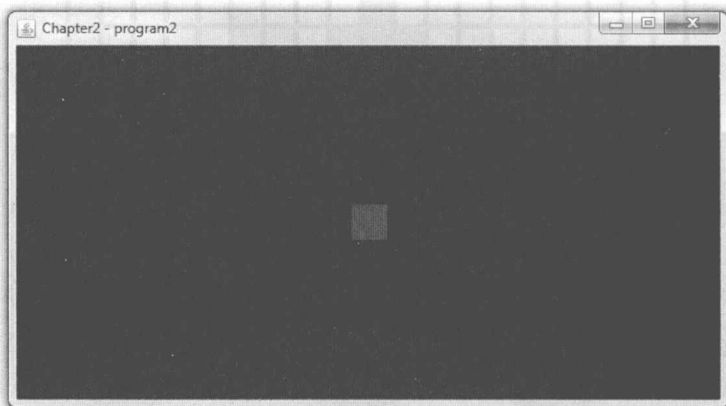


图 2.13 片段着色器颜色变化

2.1.7 像素操作

当我们在 display()方法中使用 glDrawArrays()命令绘制场景中的物体时，我们通常期望前面的物体挡住后面的物体。这也可以推广到物体自身，我们通常期望看到物体的正面对着我们，而不是背对我们。

为了实现这个效果，我们需要*隐藏面消除*（Hidden Surface Removal，HSR）。基于场景需要的不同效果，OpenGL 可以进行一系列不同的 HSR 操作。虽然这个阶段不可编程，但是理解它的工作原理是非常重要的。我们不仅需要正确地配置它，之后还需要在给场景添加阴影时对它进行进一步操作。

OpenGL 通过精巧地协调两个缓冲区完成隐藏面消除：*颜色缓冲区*（我们之前讨论过）和*深度缓冲区*（也叫作 Z 缓冲、Z-buffer）。这两个缓冲区都和光栅的大小相同——即对于屏幕上每个像素，在两个缓冲区都各有一个对应条目。

当绘制场景中的各种对象时，片段着色器会生成像素颜色。像素颜色会存放在颜色缓冲区中——颜色缓冲区最终会被写入屏幕。当多个对象占据颜色缓冲区中的相同像素时，必须根据哪个对象最接近观察者来确定保留哪个像素颜色。

隐藏面消除按照如下步骤完成。

（1）在每个场景渲染前，深度缓冲区全部初始化为表示最大深度的值。

（2）当像素颜色由片段着色器输出时，计算它到观察者的距离。

（3）如果距离小于深度缓冲区存储的值（对当前像素），那么用当前像素颜色替换颜色缓冲区中的颜色，同时用当前距离替换深度缓冲区中的值，否则抛弃当前像素。

这个过程叫作 Z-Buffer 算法，如图 2.14 所示。

```
Color [ ] [ ] colorBuf = new Color [pixelRows][pixelCols];
double [ ] [ ] depthBuf = new double [pixelRows][pixelCols];
for (each row and column)   // 初始化颜色和深度缓冲区
{   colorBuf [row][col] = backgroundColor;
    depthBuf [row][col] = far away;
}
for (each shape)   // 当新的像素更近时，更新缓冲区
{   for (each pixel in the shape)
    {   if (depth at pixel < depthBuf value)
        {   depthBuf [pixel.row][pixel.col] = depth at pixel;
            colorBuf [pixel.row][pixel.col] = color at pixel;
} } }
return colorBuf;
```

图 2.14 Z-buffer 算法

2.2 检测 OpenGL 和 GLSL 错误

编译和运行 GLSL 代码与普通编码的过程不同，GLSL 编译发生在 C++运行时。另外一个复杂的地方是 GLSL 代码并没有运行在 CPU 中（它运行在 GPU 上），因此操作系统不总是能够捕获 OpenGL 运行时的错误。以上这些使得调试变得很困难，因为常常很难检测着色器是否失败，以及为什么失败。

程序 2.3 展示了用于捕获和显示 GLSL 错误的模块。其中 GLSL 函数 glGetShaderiv()和 glGetProgramiv()用于提供有关编译过的 GLSL 着色器和程序的信息。还有之前程序 2.2 中的 createShaderProgram()函数，不过加入了错误检测的调用。

程序 2.3 包含如下 3 个实用程序。

- checkOpenGLError：检查 OpenGL 错误标志，即是否发生 OpenGL 错误。
- printShaderLog：当 GLSL 编译失败时，显示 OpenGL 日志内容。
- printProgramLog：当 GLSL 链接失败时，显示 OpenGL 日志内容。

checkOpenGLError()既用于检测 GLSL 编译错误，又用于检测 OpenGL 运行时的错误，因此我们强烈建议在整个 C++/OpenGL 应用程序开发过程中使用它。例如，在之前的程序 2.2 中，对于 glCompileShader()和 glLinkProgram()的调用很容易用程序 2.3 的代码进行加强，来确认所有的拼写错误和编译错误都能被捕获到，同时报告其原因。

用这些工具很重要的另一个原因是，GLSL 错误并不会导致 C++程序崩溃。因此，除非程序员通过步进找到错误发生的点，否则调试会非常困难。

程序 2.3 用以捕获 GLSL 错误的模块

```
void printShaderLog(GLuint shader) {
   int len = 0;
   int chWrittn = 0;
   char *log;
   glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
   if (len > 0) {
        log = (char *)malloc(len);
        glGetShaderInfoLog(shader, len, &chWrittn, log);
        cout << "Shader Info Log: " << log << endl;
        free(log);
```

```
} }

void printProgramLog(int prog) {
   int len = 0;
   int chWrittn = 0;
   char *log;
   glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
   if (len > 0) {
        log = (char *)malloc(len);
        glGetProgramInfoLog(prog, len, &chWrittn, log);
        cout << "Program Info Log: " << log << endl;
        free(log);
} }

bool checkOpenGLError() {
   bool foundError = false;
   int glErr = glGetError();
   while (glErr != GL_NO_ERROR) {
      cout << "glError: " << glErr << endl;
      foundError = true;
      glErr = glGetError();
   }
   return foundError;
}
```

检测 OpengGL 错误的示例如下：

```
GLuint createShaderProgram() {
  GLint vertCompiled;
  GLint fragCompiled;
  GLint linked;
  . . .
  // 捕获编译着色器时的错误

  glCompileShader(vShader);
  checkOpenGLError();
  glGetShaderiv(vShader, GL_COMPILE_STATUS, &vertCompiled);
  if (vertCompiled != 1) {
       cout << "vertex compilation failed" << endl;
       printShaderLog(vShader);
  }

  glCompileShader(fShader);
  checkOpenGLError();
  glGetShaderiv(fShader, GL_COMPILE_STATUS, &fragCompiled);
  if (fragCompiled != 1) {
       cout << "fragment compilation failed" << endl;
       printShaderLog(fShader);
  }

  // 捕获链接着色器时的错误

  glAttachShader(vfProgram, vShader);
  glAttachShader(vfProgram, fShader);
  glLinkProgram(vfProgram);
  checkOpenGLError();
  glGetProgramiv(vfProgram, GL_LINK_STATUS, &linked);
  if (linked != 1) {
       cout << "linking failed" << endl;
       printProgramLog(vfProgram);
  }
  return vfProgram;
}
```

还有一些其他用于推测着色器代码运行时错误成因的技巧。着色器运行时错误的常见结果是输出屏幕上完全空白，根本没有输出。即使是着色器中的一个小拼写错误也可能导致这种结果，这样就很难断定是哪个管线阶段发生了错误。没有任何输出的情况下，找到错误的成因就像大海捞针。

其中一种有用的技巧就是暂时将片段着色器换成程序 2.2 中的片段着色器。回忆程序 2.2 中，片段着色器仅输出一个特定颜色——例如蓝色。如果后来的输出中的几何形状正确（但是全是蓝色），那么顶点着色器应该是正确的，错误应该发生在片段着色器。如果输出的仍然是空白屏幕，那错误很可能发生在管线的更早期，譬如顶点着色器。

在附录 C 中，我们展示了另一种有用的调试工具，叫作 Nsight，适用于特定型号 Nvidia 显卡的机器。

2.3 从文件读取 GLSL 源代码

到此为止，GLSL 着色器代码已经内联存储在字符串中了。当程序变得更复杂时，这么做就不实际了。我们应当将我们的着色器代码存在文件中并读入它们。

读入文本文件是基础 C++技能，我们在此就不赘述了。但是，为实用起见，用于读取着色器的代码 readShaderSource()在程序 2.4 中提供。它读取着色器文本文件并返回一个字符串数组，其中每个字符串是文件中的一行文本。然后根据读入的行数确定该数组的大小。注意，createShaderProgram()在这里替换了程序 2.2 中的版本。在本例中，顶点着色器和片段着色器代码现在分别放在文本文件“vertShader.glsl”和“fragShader.glsl”中。

程序 2.4 从文件读取 GLSL 源文件

```
(....#includes 与之前相同, main(), display(), init()也与之前相同, 同时加入如下代码...)
#include <string>
#include <iostream>
#include <fstream>
...
string readShaderSource(const char *filePath) {
    string content;
    ifstream fileStream(filePath, ios::in);
    string line = "";
    while (!fileStream.eof()) {
        getline(fileStream, line);
        content.append(line + "\n");
    }
    fileStream.close();
    return content;
}

GLuint createShaderProgram() {
    (...与之前相同, 同时加入如下代码...)
    string vertShaderStr = readShaderSource("vertShader.glsl");
    string fragShaderStr = readShaderSource("fragShader.glsl");

    const char *vertShaderSrc = vertShaderStr.c_str();
    const char *fragShaderSrc = fragShaderStr.c_str();
```

```
    glShaderSource(vShader, 1, &vertShaderSrc, NULL);
    glShaderSource(fShader, 1, &fragShaderSrc, NULL);

    (…构建如前的渲染程序)
}
```

2.4 从顶点构建对象

最终我们想要绘制的不止是一个单独的点，而是想要绘制由很多顶点组成的对象。本书的大部分章节将会致力于这一主题。现在我们从一个简单的例子开始——我们将会定义 3 个顶点，并用它们绘制一个三角形，如图 2.15 所示。

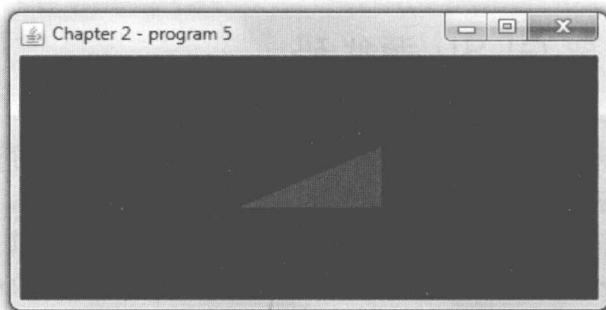


图 2.15 绘制简单三角形

我们可以通过对程序 2.2（事实上是从文件读入着色器的程序 2.4）进行两个小改动来实现绘制三角形：(a) 修改顶点着色器，以便将 3 个不同的点输出到后续的管线阶段；(b) 修改 glDrawArrays()调用，指定 3 个顶点。

在 C++/OpenGL 应用程序中［特别是在 glDrawArrays() 调用中］我们指定了 GL_TRIANGLES（而非 GL_POINTS），同时也指定了管线中有 3 个顶点。这样顶点着色器会在每个迭代运行 3 遍，内置变量 gl_VertexID 会自增（初始值为 0）。通过检测 gl_VertexID 的值，着色器设计为可以在每次运行时输出不同的点。前面说到这 3 个点之后会经过光栅化阶段，生成一个填充过的三角形。程序的改动显示在程序 2.5 中（余下的代码与之前在程序 2.4 中的相同）。

程序 2.5 绘制三角形

顶点着色器

```
#version 430
void main(void)
{ if (gl_VertexID == 0) gl_Position = vec4( 0.25, -0.25, 0.0, 1.0);
  else if (gl_VertexID == 1) gl_Position = vec4(-0.25, -0.25, 0.0, 1.0);
  else gl_Position = vec4( 0.25, 0.25, 0.0, 1.0);
}
```

C++/OpenGL 应用程序——在 display()函数中

```
. . .
glDrawArrays(GL_TRIANGLES, 0, 3);
```

2.5 场景动画

本书中的很多技术可以用于动画。当场景中的物体移动或改变时，场景会被重复渲染以实时反映这些改动。

回顾 2.1.1 小节中，我们构建的 main()函数只调用了 init()一次，之后就重复调用 display()。因此虽然前面所有的例子看起来都是静态绘制的场景，但实际上 main()函数中的循环会让它们一次又一次地绘制。

因此，main()函数的结构已经可以支持动画了。我们只需要设计 display()函数来随时间改变绘制的东西。场景的每一次绘制都叫作一帧，调用 display()的频率叫作帧率。在程序逻辑中移动的速率可以通过自前一帧到目前经过的时间来控制（这就是为什么我们会将“currentTime”作为 display()函数的参数）。

程序 2.6 中展示了动画示例。我们使用了程序 2.5 中的三角形，并给它加入了先向右，再向左，往复移动的动画。在本例中，我们不考虑经过的时间，因此三角形的移动或快或慢，基于运行计算机的处理速度。在未来的示例中，我们将会使用经过的时间来确保无论在什么配置的计算机上运行，动画都保持以同样的速度播放。

在程序 2.6 中，程序的 display()方法维持一个变量“*x*”用于偏移三角形的 *X* 轴位置。每当 display()调用时，它的值都会改变（因此每帧都不同）。同时每当它到达 1.0 或者−1.0 时，都会改变方向。在 *x* 中的值会被复制到顶点着色器的“offset”变量中。执行这个复制的机制叫作 Uniform 变量（统一变量），稍后我们会在第 4 章中学习它。目前不必了解统一变量的细节。现在，只需要注意 C++/OpenGL 应用程序先调用 glGetUniformLocation()获取指向“offset”变量的指针，之后调用 glProgramUniform1f()将 *x* 的值复制给 offset。之后顶点着色器会将 offset 加给所绘制三角形的 *X* 坐标。注意，每次调用 display()时背景都会被清除，以避免三角形移动时留下一串轨迹。图 2.16 展示了 3 个时间点显示的图像（当然，书中的静态图是无法展示移动的）。

程序 2.6　简单动画示例

C++/OpenGL 应用程序：

```
// #includes 与之前相同，定义也与之前相同，同时加入如下代码：
  float x = 0.0f;            // 三角形在 x 轴的位置
  float inc = 0.01f;         // 移动三角形的偏移量

  void display(GLFWwindow* window, double currentTime) {
     glClear(GL_DEPTH_BUFFER_BIT);
     glClearColor(0.0, 0.0, 0.0, 1.0);
     glClear(GL_COLOR_BUFFER_BIT);      // 每次将背景清除为黑色

     glUseProgram(renderingProgram);

     x += inc;                          // 切换至让三角形向右移动
     if (x > 1.0f) inc = -0.01f;        // 沿 x 轴移动三角形
     if (x < -1.0f) inc = 0.01f;        // 切换至让三角形向左移动
     GLuint offsetLoc = glGetUniformLocation(renderingProgram, "offset");    // 获取 "offset" 指针
```

```
        glProgramUniform1f(renderingProgram, offsetLoc, x);      // 将 "x" 中的值传给 "offset"

        glDrawArrays(GL_TRIANGLES,0,3);
    }
    . . . // 其余函数同前例
}
```

顶点着色器：

```
#version 430
uniform float offset;
void main(void)
{ if (gl_VertexID == 0) gl_Position = vec4( 0.25 + offset, -0.25, 0.0, 1.0);
  else if (gl_VertexID == 1) gl_Position = vec4(-0.25 + offset, -0.25, 0.0, 1.0);
  else gl_Position = vec4( 0.25 + offset, 0.25, 0.0, 1.0);
}
```

注意，除了添加三角形动画代码之外，我们还在 display()函数的开头添加了这行代码：

```
glClear(GL_DEPTH_BUFFER_BIT);
```

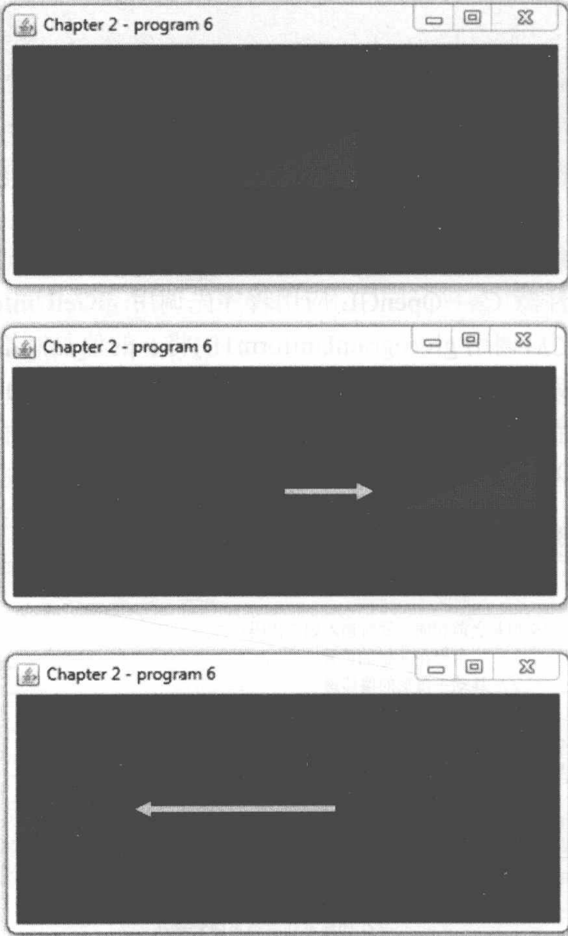


图 2.16　移动的三角形动画

虽然在本例中并不是必需的，我们仍然把它加在这里，同时它会在之后的大多数应用程

序中存在。回忆 2.1.7 小节中讨论的，隐藏面消除需要同时用到颜色缓冲区和深度缓冲区。当我们后面渐渐地开始绘制更复杂的 3D 场景时，每帧初始化（清除）深度缓冲区就是必要的，尤其是对于动画场景，要确保深度对比不会受旧的深度数据影响。从前面的例子中可以明显看出，清除深度缓冲区的命令与清除颜色缓冲区的命令基本相同。

2.6 C++代码文件结构

目前为止，我们的所有 C++/OpenGL 应用程序代码都放在同一个叫作“main.cpp”的文件中，GLSL 着色器代码放在“vertShader.glsl”和“fragShader.glsl”文件中。我们承认在 main.cpp 中塞进很多应用代码不是最佳实践，但我们在本书中采用这个约定，以便于在每个例子中，哪个文件包含这个例子中主要的 C++/OpenGL 代码这件事都很清楚。在本教材中，主要的 C++/OpenGL 文件总是叫作“main.cpp”。在实践中，应用程序当然应该模块化，以适当对应应用的各功能。

但是，当我们继续学习时，我们会遇到一些情况。在这些情况下，我们会创建一些实用的模块，并在不同的应用程序中使用。当时机适当，我们会将这些模块分离到单独的文件中以便重用。例如，稍后我们会定义一个 Sphere 类。这个类会在很多例子中用到，因此它会分到它自己的文件（Sphere.cpp 和 Sphere.h）中。

相似地，当我们遇到需要重用函数的时候，我们会把它们放进“Utils.cpp”（与“Utils.h”关联）。我们已经看到好几个适合放进“Utils.cpp”的函数了：2.2 节中描述的错误检测模块和 2.3 节中描述的用来读入 GLSL 着色器的函数。后者非常适合重载，如“createShaderProgram()”可以对应用中所有可能的管线着色器组合进行定义：

- `GLuint Utils::createShaderProgram(const char *vp, const char *fp)`
- `GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp)`
- `GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES, const char *fp)`
- `GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES, const char *gp, const char *fp)`

以上列出的第一个条目支持仅使用了顶点着色器和片段着色器的程序。第二个支持使用了顶点着色器、几何着色器和片段着色器的情况。第三个支持用了顶点着色器、曲面细分着色器和片段着色器的情况。第四个支持用了顶点着色器、曲面细分着色器、几何着色器和片段着色器的情况。每个条目中，接受的参数都包含着色器代码的 GLSL 文件路径。例如，如下调用使用了其中一个重载函数，以编译并链接包含顶点着色器和片段着色器的管线。编译链接后的程序被放在变量“renderingProgram”中：

```
renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");
```

这些 createShaderProgram()实现都可以在随书附赠的配套资源中找到（在“Utils.cpp”文件中），同时它们都包含了 2.2 节中的错误处理。它们并没有什么新内容，只是用这种方式组织以便使用。随着我们继续向前推进本书，会有更多相似的函数加入 Utils.cpp。我们强烈鼓励读者阅读配套资源中的 Utils.cpp 文件，甚至有需要时可在其中加入函数。配套资源中的

程序是根据学习本书的方法构建的，因此了解它们的结构应当有助加强自己对书中内容的理解。

我们对于在“Utils.cpp”文件中的函数，都以静态函数进行实现，因此不需要实例化 Utils 类。基于正在开发的系统架构，读者可能会倾向于使用实例方法甚至独立函数实现它们。

我们所有的着色器文件都使用“.glsl”后缀。

补充说明

在本章中，还有很多我们没有讨论到的 OpenGL 管线细节。我们略过了许多内部阶段，同时完全省略了纹理的处理。我们在本章的目标是，对后面要用来编码的框架有尽可能简单的整体印象。当我们继续学习时，会学到更多的细节。同时我们也推迟了展示曲面细分着色器和几何着色器的代码。在之后的章节中，我们会构建一套完整的系统，来展现如何为每个阶段编写实际的着色器。

对于如何组织场景动画代码，尤其是线程管理，有着更复杂的方法。有的语言中的库，如 JOGL 和 LWJGL（对于 Java）会提供一些支持动画的类。我们鼓励对于设计特定应用渲染循环（或者“游戏循环”）感兴趣的读者去读一些在游戏引擎设计上更加专业的图书[NY14]，同时跟踪在 gamedev.net [GD17]上的讨论。

我们在 glShaderSource()命令上注释了一个细节。它的第四个参数指定了一个“长度数组”，其中包括给定着色器程序中每行代码的字符串的整数长度。如果这个参数被设为 null，像我们之前那样，OpenGL 将会自动从以 null 结尾的字符串中构建这个数组。因此我们特地确保所有我们传给 glShaderSource()的字符串都是以 null 结尾的[通过在 createShaderProgram()中调用 c_str()函数]。实际中通常也会遇到手动构建这些数组而非传入 null 的应用程序。

在本书中，读者可能多次想要了解 OpenGL 某些方面的数值限制。例如，程序员可能需要知道几何着色器可以生成的最大输出数，或者可以为渲染点指定的最大尺寸。这些值中很多都依赖于实现，即在不同的机器上是不同的。OpenGL 提供了通过使用 glGet()指令来获取这些值的机制。基于查询的参数的不同类型，glGet()指令也有着不同的形式。例如，查询点的尺寸的最大值时，如下调用会将最小值和最大值（基于运行机器上的 OpenGL 实现）放入名为“size”的数组中的前两个元素。

```
glGetFloatv(GL_POINT_SIZE_RANGE, size)
```

这类查询有很多。更多示例参见 OpenGL 参考文档[OP16]。

在本章中，我们尝试在每次 OpenGL 调用时，描述其各个参数。当我们向后推进时，这么做就会显得冗余，因此当我们觉得描述参数只会妨碍理解时，就不会描述该参数。这是因为很多 OpenGL 函数有大量与我们示例无关的参数。必要时读者应当使用 OpenGL 文档来获取参数详情。

习题

2.1 修改程序 2.2，增加动画，让绘制的点周而复始地放大和缩小。提示：使用 glPointSize()

方法，用一个变量作为参数。

2.2 修改程序 2.5，使之绘制等腰三角形（而非图 2.15 所示的直角三角形）。

2.3 （项目）修改程序 2.5，使之包含程序 2.3 中所示的错误检查模块。之后，尝试在着色器中加入各种错误，同时观察渲染行为以及生成的错误信息。

参考资料

[GD17] Game Development Network, accessed October 2018.

[NY14] R. Nystrom, "Game Loop," in *Game Programming Patterns* (Genever Benning, 2014), and accessed October 2018.

[OP16] OpenGL 4.5 Reference Pages, accessed July 2016.

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

《OpenGL 超级宝典（第 5 版）》（ISBN 978-7-115-27456-4，定价 108 元），俗称 OpenGL 蓝宝书。

第 3 章 数学基础

计算机图形学中大量使用了数学原理，尤其是矩阵和矩阵代数。虽然我们倾向于认为 3D 图形编程是最现代的技术领域之一（它在很多方面确实是），但它用到的很多技术实际上可以追溯到上百年前。其中一些甚至是文艺复兴时期的伟大哲学家们就已经理解并记录的。

3D 图形学中几乎每个方面、每种效果——移动、缩放、透视、纹理、光照、阴影等——都在很大程度上以数学方式实现。

这里，我们假设读者具备基础的矩阵运算知识。对于基础矩阵代数的完整讲解超出了本书的范围。因此，如果读者在任何时候发现自己不理解特定的矩阵操作，则应当先找一些相关材料进行阅读，确保完全理解矩阵操作之后再继续学习。

3.1 3D 坐标系统

3D 空间通常用 3 个坐标轴 *X*、*Y* 和 *Z* 来表示。这 3 个轴可以以两种方式来布置：*左手*或*右手*（它们是以轴的朝向来命名的，通过左手或右手大拇指与食指、中指成直角来进行构造）。[①] 如图 3.1 所示。

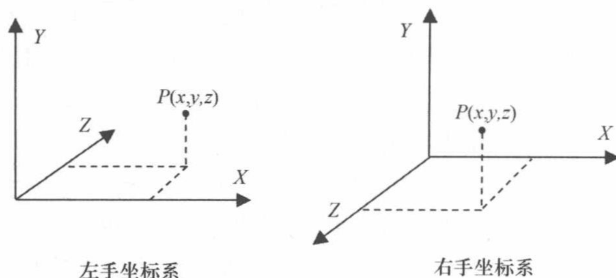


图 3.1 3D 坐标系统

知道图形编程环境所使用的坐标系是很重要的。例如，OpenGL 中的坐标系大体是右手坐标系，而 Direct3D 中大体是左手坐标系。在本书中，除非特别说明，否则我们都是用右手坐标系。

3.2 点

3D 空间中的点可以通过使用形如（2, 8, −3）的符号，列出 *X*、*Y*、*Z* 的值来表示。不过，

① 大拇指代表 *Z* 轴，食指代表 *X* 轴，中指代表 *Y* 轴。图 3.1 中，左手大拇指向外，手心向上；右手大拇指向内，手心向上。——译者注

如果用齐次坐标——一种在 19 世纪初首次描述的表示法来表示点会更有用。在每个点的齐次坐标有 4 个值。前 3 个值表示 *X*、*Y* 和 *Z*，第四个值 *W* 总是非零值，通常为 1。因此，我们会将之前的点表示为（2, 8, −3, 1）。正如我们稍后将要看到的，齐次坐标将会使我们的图形学计算更高效。

用来存储齐次 3D 坐标的 GLSL 数据类型是 vec4（“vec”代表向量，同时也可以用来表示点）。GLM 库包含适合在 C++/OpenGL 应用中创建和存储 3 元和 4 元（齐次）点的类，分别叫作 vec3 和 vec4。

3.3 矩阵

矩阵是矩形的值阵列，它的元素通常使用下标访问。第一个下标表示行号，第二个下标表示列号，下标从 0 开始。我们在 3D 图形计算中要用到的矩阵大多数大小为 4×4，如图 3.2 所示。

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

图 3.2　4×4 矩阵

GLSL 语言中的 mat4 数据类型用来存储 4×4 矩阵。同样，GLM 中有 mat4 类用以实例化并存储 4×4 矩阵。

单位矩阵中一条对角线的值为 1，其余值全为 0：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

任何值乘以单位矩阵都不会改变。在 GLM 中，调用构造函数 glm::mat4 m(1.0f)以在变量 *m* 中生成单位矩阵。

矩阵转置的计算是通过交换矩阵的行和列完成的。例如：

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{10} & A_{20} & A_{30} \\ A_{01} & A_{11} & A_{21} & A_{31} \\ A_{02} & A_{12} & A_{22} & A_{32} \\ A_{03} & A_{13} & A_{23} & A_{33} \end{bmatrix}^{\mathrm{T}}$$

GLM 库和 GLSL 库都有转置函数，分别是 glm::transpose(mat4)和 transpose(mat4)。

矩阵加法简单明了：

$$\begin{bmatrix} A+a & B+b & C+c & D+d \\ E+e & F+f & G+g & H+h \\ I+i & J+j & K+k & L+l \\ M+m & N+n & O+o & P+p \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} + \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

在 GLSL 中，+运算符在 mat4 上进行了重载，以支持矩阵加法。

3D 图形学中有很多有用的矩阵乘法操作。矩阵乘法一般可以从左向右或从右向左处理（注意，由于左乘和右乘是不同的，所以矩阵乘法不满足交换律）。

在 3D 图形学中，点与矩阵相乘通常从右向左，得到点，如：

$$\begin{pmatrix} AX+BY+CZ+D \\ EX+FY+GZ+H \\ IX+JY+KZ+L \\ MX+NY+OZ+H \end{pmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

注意，我们用齐次坐标将点(X, Y, Z)表示为列数为 1 的矩阵。

GLSL 和 GLM 都支持点（确切地说是 vec4）与矩阵使用*操作符相乘。

4×4 矩阵与 4×4 矩阵相乘如下：

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \times \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} =$$

$$\begin{bmatrix} Aa+Be+Ci+Dm & Ab+Bf+Cj+Dn & Ac+Bg+Ck+Do & Ad+Bh+Cl+Dp \\ Ea+Fe+Gi+Hm & Eb+Ff+Gj+Hn & Ec+Fg+Gk+Ho & Ed+Fh+Gl+Hp \\ Ia+Je+Ki+Lm & Ib+Jf+Kj+Ln & Ic+Jg+Kk+Lo & Id+Jh+Kl+Lp \\ Ma+Ne+Oi+Pm & Mb+Nf+Oj+Pn & Mc+Ng+Ok+Po & Md+Nh+Ol+Pp \end{bmatrix}$$

矩阵相乘也经常叫作合并，稍后我们会看到，它可以用于将一系列矩阵变换合并成一个矩阵。这种合并矩阵变换的能力来自矩阵乘法的结合律。

考虑如下运算序列：

$$\text{New Point} = \text{Matrix}_1 \times (\text{Matrix}_2 \times (\text{Matrix}_3 \times \text{Point}))$$

我们将一个点与 Matrix_3 相乘，之后将结果与 Matrix_2 相乘，最后将结果与 Matrix_1 相乘。其结果是一个新的点。结合律确保了之前的计算与如下计算相同：

$$\text{New Point} = (\text{Matrix}_1 \times \text{Matrix}_2 \times \text{Matrix}_3) \times \text{Point}$$

我们先将 3 个矩阵相乘，建立 Matrix_1、Matrix_2、Matrix_3 的连接。如果我们称其为 Matrix_C，我们就可以将之前的运算写作：

$$\text{New Point} = \text{Matrix}_C \times \text{Point}$$

我们稍后在第 4 章会看到这么做的好处是，我们需要经常将相同的一系列矩阵变换应用到场景中的每个点上。通过预先一次计算好这些矩阵的合并，就可以成倍减少总的矩阵运算量。

GLSL 和 GLM 都支持使用重载后的*运算符进行矩阵乘法。

一个 4×4 矩阵的逆矩阵是另一个 4×4 矩阵，用 $\boldsymbol{M}^{-1}$ 表示，在矩阵乘法中有如下性质：

$$\boldsymbol{M} \times \boldsymbol{M}^{-1} = \boldsymbol{M}^{-1} \times \boldsymbol{M} = \text{单位矩阵}$$

在此我们就不展示计算逆矩阵的细节了。但是，需要知道的是计算矩阵的逆矩阵的运算量很大。幸运的是我们只有很少情况下需要用到它。在这些极少的情况下，GLSL 和 GLM 都提供了 mat4.inverse()函数。

3.4 变换矩阵

在图形学中，矩阵通常用来进行物体的变换。例如矩阵可以用来将点从一处移动到另一处。在本章中，我们将会学习 5 个有用的变换矩阵：

- 平移矩阵；
- 缩放矩阵；
- 旋转矩阵；
- 投影矩阵；
- LookAt 矩阵。

变换矩阵的重要特性之一就是它们都是 4×4 矩阵。这是因为我们决定使用齐次坐标系。否则，各变换矩阵可能会有不同的维度并且无法相乘。正如我们所见，确保变换矩阵大小相同并不只是为了方便，同时让它们可以任意组合，进行预先计算变换矩阵以提升性能。

3.4.1 平移矩阵

平移矩阵用于将物体从一个位置移至另一位置。它包含一个单位矩阵，同时 X、Y 和 Z 的移动量在 A_{03}、A_{13}、A_{23}。图 3.3 展示了平移矩阵和它与齐次坐标点相乘的效果。其结果是一个以平移值"移动过"的点。

$$\begin{pmatrix} X + T_X \\ Y + T_Y \\ Z + T_Z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_X \\ 0 & 1 & 0 & T_Y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

图 3.3 平移矩阵变换

注意，作为与平移矩阵相乘的结果，点(X, Y, Z)平移(或移动)到了位置$(X+T_x, Y+T_y, Z+T_z)$。同样需要注意的是这个乘法是从右向左相乘。

例如，当我们想要将一组点向上沿 Y 轴正方向移动 5 个单位，我们可以通过给一个单位矩阵的 T_y 位置放入 5 来构建平移矩阵。之后我们只需要将我们想要移动的点与矩阵相乘就可以了。

GLM 中有一些函数是用于构建与点相乘的平移矩阵的。其中相关的操作有：

- glm::translate(x, y, z)构建平移(x, y, z)的矩阵；
- mat4 × vec4。

3.4.2 缩放矩阵

缩放矩阵用于改变物体的大小或者将点向原点相反方向移动。虽然缩放点这个操作乍一看有点奇怪，不过 OpenGL 中的物体都是用一组或多组的点定义的。因此，缩放物体涉及缩放它的点的集合。

缩放矩阵变换由单位矩阵和位于 A_{00}, A_{11}, A_{22} 的 X、Y、Z 缩放因子组成。图 3.4 中展示了缩放矩阵的形式和当它与齐次坐标点相乘的效果：所得的结果是经过缩放值修改后的新点。

$$\begin{pmatrix} X * S_X \\ Y * S_Y \\ Z * S_Z \\ 1 \end{pmatrix} = \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

图 3.4 缩放矩阵变换

GLM 中有一些函数是用于构建与点相乘的缩放矩阵的。其中相关的操作有：

- glm::scale(x, y, z) 构建缩放(x, y, z)的矩阵；
- mat4 × vec4。

缩放还可以用来切换坐标系。例如，我们可以用缩放来在给定右手坐标系的情况下确定左手坐标。从图 3.1 中我们可以看到通过反转 Z 坐标就可以在右手坐标系和左手坐标系中切换，因此，用来切换坐标系的缩放矩阵变换是：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4.3 旋转矩阵

旋转会稍微复杂一些，因为在 3D 空间中旋转物体需要指定旋转轴和旋转的角度或弧度。

在 16 世纪中叶，数学家莱昂哈德・欧拉表明，围绕任何轴的旋转都可以表示为绕 X、Y、Z 轴旋转的组合[EU76]。围绕这 3 个轴的旋转角度被称为欧拉角。这个被称为欧拉定理的发现，对我们很有用，因为对于每个坐标轴的旋转可以用矩阵变换来表示。

旋转变换有 3 种，分别是绕 X、Y 和 Z 轴旋转，见图 3.5。同时 GLM 中也有一些用于构建旋转矩阵的函数。

- glm::rotate(mat4, θ, x, y, z)构建绕 X, Y, Z 轴旋转 θ 度的缩放矩阵。
- mat4 × vec4。

绕X轴旋转θ度：

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

绕Y轴旋转θ度：

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

绕Z轴旋转θ度：

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

图 3.5 旋转变换矩阵

实践中，当 3D 空间中旋转轴不穿过原点时，物体使用欧拉角进行旋转需要几个额外的步骤。一般有：

（1）平移旋转轴以使它经过原点；

（2）绕 X、Y 和 Z 轴旋转适当的欧拉角；

（3）复原步骤（1）中的平移。

图 3.5 中所示的 3 个旋转变换都有自己有趣的特性，即反向旋转的矩阵恰等于其转置矩阵。通过观察之前这些矩阵，同时有 $\cos(-\theta) = \cos(\theta)$ 和 $\sin(-\theta) = -\sin(\theta)$，即可验证。后面将会用到这个特性。

欧拉角在某些 3D 图形应用中会导致一些瑕疵。因此，通常在计算旋转时推荐使用四元数。有兴趣探索四元数的读者可以寻求很多已有的资源（如[KU98]）。欧拉角足以满足我们的大部分需求。

3.5 向量

向量表示大小和方向。它们没有特定位置。“移动”向量并不改变它所代表的含义。

记录向量的方法各式各样，如：一端带箭头的线段、二元组（幅度，方向）或两点之差。在 3D 图形学中，向量一般用空间中的单个点表示，向量的大小是原点到该点的距离，方向则是原点到该点的方向。在图 3.6 中，向量 $\boldsymbol{V}$ 可以用点 P_1 和 P_2 之间的差表示，也可以等价地用原点到 P_3 来表示。在我们的所有应用中，我们都简单地将 $\boldsymbol{V}$ 表示为(x, y, z)，即我们用来表示 P_3 的符号。

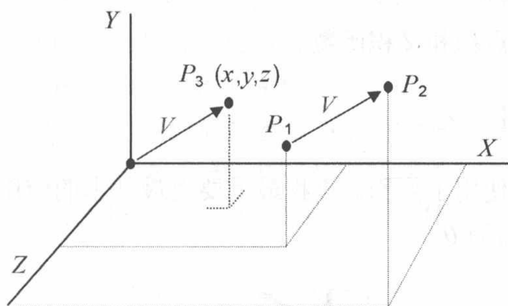


图 3.6 向量 $\boldsymbol{V}$ 的两种表示

用与表示点相同的方式来表示向量很方便，因为我们对点和向量用同样的矩阵变换。不过这也会使人困惑。因此，我们有时候会在向量上加一个小箭头（如 $\vec{V}$ ）。许多图形系统并不区分点和向量，如 GLSL 和 GLM，它们所提供的 vec3/vec4 类型既能用来存储点，又能用来存储向量。有的系统（例如本书 Java 早期版本中所用到的 graphicslib3D 库）对于点和向量有着不同的类，强制使用适当的类来进行所需的操作。对于点和向量使用同一种类型还是不同的类型哪个更好这件事上仍然没有定论。

在 GLM 和 GLSL 中有许多 3D 图形学中经常用到的向量操作。如假设有向量 $\boldsymbol{A}(u, v, w)$ 和 $\boldsymbol{B}(x, y, z)$：

加减法：

$\boldsymbol{A} \pm \boldsymbol{B} = (u \pm x, v \pm y, w \pm z)$

glm: vec3 ± vec3

GLSL: vec3 ± vec3

归一化（变为长度 =1）：

$\hat{\boldsymbol{A}} = \boldsymbol{A}/|\boldsymbol{A}| = \boldsymbol{A}/\mathrm{sqrt}(u^2+v^2+w^2)$，其中 $|\boldsymbol{A}| \equiv$ 向量 $\boldsymbol{A}$ 的长度

glm: normalize(vec3) 或 normalize(vec4)

GLSL: normalize(vec3) 或 normalize(vec4)

点积：

$\boldsymbol{A} \cdot \boldsymbol{B} = ux + vy + wz$

glm: dot(vec3,vec3) 或 dot(vec4,vec4)

GLSL: dot(vec3,vec3) 或 dot(vec4,vec4)

叉积：

$\boldsymbol{A} \times \boldsymbol{B} = (vz\text{-}wy, wx\text{-}uz, uy\text{-}vx)$

glm: cross(vec3,vec3)

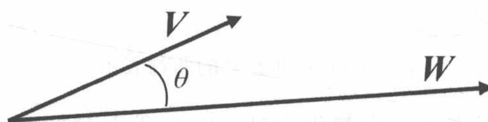
GLSL: cross(vec3,vec3)

其他有用的向量函数如 magnitude（在 GLSL 和 GLM 中是 length()）、reflection 和 refraction（在 GLSL 和 GLM 中都有）。

我们现在仔细看一下点积和叉积函数。

3.5.1 点积的应用

在本书中的程序大量使用了点积。点积最重要也最基本的应用是求解两向量夹角。设有向量 $\boldsymbol{V}$ 和 $\boldsymbol{W}$，计算其夹角为 θ。



$$\boldsymbol{V} \bullet \boldsymbol{W} = |\boldsymbol{V}||\boldsymbol{W}|\cos(\theta)$$

$$\cos(\theta) = \frac{\boldsymbol{V} \bullet \boldsymbol{W}}{|\boldsymbol{V}||\boldsymbol{W}|}$$

因此，如果 $\boldsymbol{V}$ 和 $\boldsymbol{W}$ 是正规化向量（有着单位长度的向量——这里用“^”标记正规化），则有：

$$\cos(\theta) = \hat{\boldsymbol{V}} \bullet \hat{\boldsymbol{W}}$$

$$\theta = \arccos(\hat{\boldsymbol{V}} \bullet \hat{\boldsymbol{W}})$$

有趣的是，我们后面会看到通常用到的是 cos(θ)，而非 θ。因此，这两个推导出的公式都很有用。

点积同时还有许多其他用途。

- 求解向量的大小：$\sqrt{\boldsymbol{V} \bullet \boldsymbol{V}}$。
- 求解两向量是否正交，若正交，则 $\boldsymbol{V} \bullet \boldsymbol{W} = 0$。
- 求解两向量是否平行，若平行，则 $\boldsymbol{V} \bullet \boldsymbol{W} = |\boldsymbol{V}||\boldsymbol{W}|$。
- 求解两向量是否平行但指向相反方向，若满足，则 $\boldsymbol{V} \bullet \boldsymbol{W} = -|\boldsymbol{V}||\boldsymbol{W}|$。
- 求解两向量夹角是否在（−90°～+90°）：$\hat{\boldsymbol{V}} \bullet \hat{\boldsymbol{W}} > 0$。
- 求解点 $P=(x, y, z)$到平面 $S=(a, b, c, d)$的最小有符号距离。首先，求垂直于 S 的单位法向量：$\hat{\boldsymbol{n}} = \left(\frac{a}{\sqrt{a^2+b^2+c^2}}, \frac{b}{\sqrt{a^2+b^2+c^2}}, \frac{c}{\sqrt{a^2+b^2+c^2}} \right)$，以及从原点到平面的最短距离 $D = \frac{d}{\sqrt{a^2+b^2+c^2}}$。之后从 P 到 S 的最小有符号距离为 $(\hat{\boldsymbol{n}} \bullet P) + D$，其符号由 P 在 S 的哪边决定。

3.5.2 叉积的应用

两向量叉积的一个重要特性是，它会生成一个新的向量，新的向量正交（垂直）于之前两个向量所定义的平面。我们会在本书中大量使用到这一特性。

任意两个不共线向量都定义了一个平面。例如考虑两个任意向量 $\boldsymbol{V}$ 和 $\boldsymbol{W}$。由于向量可以在不改变含义的情况下进行平移，因此，可以将它们移动到起点相交的位置。图 3.7 展示了 $\boldsymbol{V}$ 和 $\boldsymbol{W}$ 定义的平面，以及其叉积所得法向量。其所得法向量的方向遵循**右手定则**，即将右手手指从 $\boldsymbol{V}$ 向 $\boldsymbol{W}$ 卷曲会使得大拇指指向法向量 $\boldsymbol{R}$。

注意，这里顺序很重要。$\boldsymbol{W} \times \boldsymbol{V}$ 将会得到与 $\boldsymbol{R}$ 方向相反的向量。

通过叉积来获得法向量的能力对我们后面要学习的光照部分非常重要。为了确定光照效果，我们需要知道所渲染模型的**外向法向量**。图 3.8 中展示了一个例子，其中有一个 6 个点（顶点）构成的简单模型，使用叉积计算来获得其中一面的外向法向量。

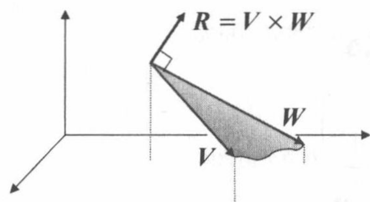


图 3.7 叉积得到法向量

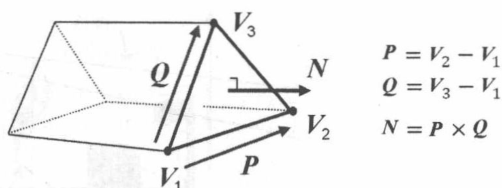


图 3.8 计算外向法向量

3.6 局部和世界空间

3D 图形学（用 OpenGL 或其他框架）常见的应用是模拟三维世界、在其中放入物体并

在显示器上观看它。放在3D世界的物体通常用三角形的集合来进行建模。稍后我们会在第6章中详细讲解建模。但是我们可以先了解一下大致的处理过程。

当建立物体的3D模型时，我们通常以最方便的定位方式描述模型。如果模型是个球形，那么我们很可能将球心定位于原点（0,0,0）并赋予它一个方便的半径，比如1。模型定义的空间叫作局部空间（local space）或模型空间（model space）。OpenGL文档使用的术语是物体空间（object space）。

之后这个球形可能用于一个大模型的部分，如成为机器人的头部。这个机器人，当然，定义在它自己的局部/模型空间。我们可以用图3.9所示的矩阵变换通过缩放、旋转和平移，将球形模型放在机器人模型的空间。通过这种方式，可以分层次地构建复杂模型（在4.8节中会进一步通过使用一堆矩阵讲解这个主题）。

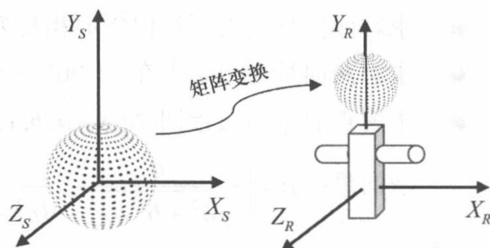


图3.9 球形和机器人的模型空间

使用同样的方式，通过设定物体在模拟世界中的朝向和大小，将物体放在模拟这个世界的空间中，这个空间叫作世界空间。将对象定位及定向在世界空间的矩阵称为模型矩阵或 $\boldsymbol{M}$。

3.7 视觉空间和合成相机

到此为止，我们所接触的变换矩阵全都在3D空间中操作。但是，我们最终需要将3D空间——或它的一部分——展示在2D显示器上。为了达成这个目标，我们需要找到一个有利点。正如我们在现实世界通过眼睛从一点观察一样，我们也必须找到一点并确立观察方向作为我们观察虚拟世界的窗口。这个点叫作“视图”或“视觉”空间，或“合成相机”。

如图3.10和图3.12所示，观察3D世界需要：（a）将相机放入世界的某个位置；（b）调整相机的角度，通常需要一套它自己的直角坐标轴 $U/V/N$；（c）定义一个视体（view volume）；（d）将视体内的对象投影到投影平面（projection plane）上。

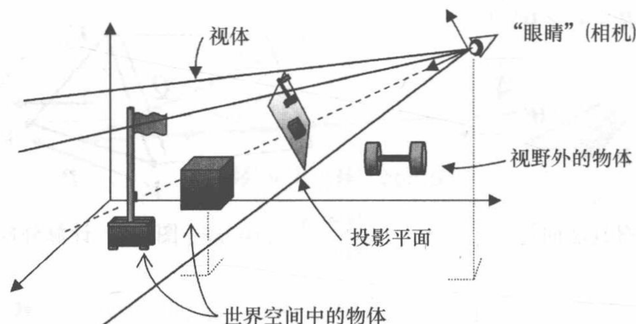


图3.10 将相机放入3D世界中

OpenGL有一个固定在原点(0,0,0)并向下看向 Z 轴负方向的相机，如图3.11所示。

为了应用OpenGL相机，我们需要做的是将它模拟移动到适合的位置和方向。我们需要

先找出在世界中的物体与我们期望的相机位置的相对位置（如物体应该在由图 3.12 所示相机 U、V、N 轴定义的“相机空间”中的何处）。给定世界空间中的点 P_W，我们需要通过变换将它转换成相应相机空间中的点，从而让它看起来好像是我们从期望的相机位置 C_W 进行观看的。我们通过计算它在相机空间中的位置 P_C 实现。已知 OpenGL 相机位置永远固定在点(0,0,0)，问：我们如何变换来实现上述功能？

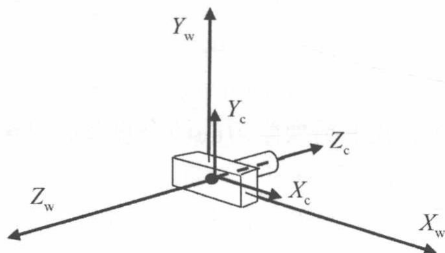


图 3.11 OpenGL 固定相机

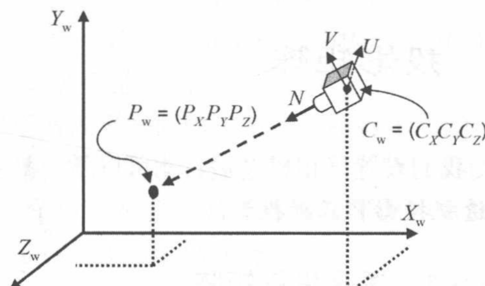


图 3.12 相机方向

需要做的变换如下：

（1）将 P_W 平移，其向量为负的期望相机位置。

（2）将 P_W 旋转，其角度为负的期望相机旋转的欧拉角。

我们可以构建一个单一变换矩阵以完成旋转和平移，这个矩阵叫作视图变换（viewing transform）矩阵 $\boldsymbol{V}$。矩阵 $\boldsymbol{V}$ 通过合并矩阵 $\boldsymbol{T}$（包含负相机期望位置的平移矩阵）和 $\boldsymbol{R}$（包含负相机期望旋转的旋转矩阵）。在本例中，从右向左，我们先平移世界空间中的点 P_W，之后旋转：

$$P_C = \boldsymbol{R}\,(\,\boldsymbol{T} * P_W\,)$$

如前所见，通过结合律我们得到如下运算：

$$P_C = (\,\boldsymbol{R} * \boldsymbol{T}\,)\,P_W$$

如果我们将合并后的 $\boldsymbol{R} * \boldsymbol{T}$ 存入矩阵 $\boldsymbol{V}$，运算成为

$$P_C = \boldsymbol{V} * P_W$$

完整的计算以及 $\boldsymbol{T}$ 和 $\boldsymbol{R}$ 的准确内容在图 3.13 中（我们忽略了对矩阵 $\boldsymbol{R}$ 的推导——推导过程见参考资料[FV95]）。

负相机旋转角　　负相机位置

$$\begin{pmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{pmatrix} = \underbrace{\underbrace{\begin{bmatrix} \widehat{U}_X & \widehat{U}_Y & \widehat{U}_Z & 0 \\ \widehat{V}_X & \widehat{V}_Y & \widehat{V}_Z & 0 \\ \widehat{N}_X & \widehat{N}_Y & \widehat{N}_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{R(\text{旋转})} \underbrace{\begin{bmatrix} 1 & 0 & 0 & -C_X \\ 0 & 1 & 0 & -C_Y \\ 0 & 0 & 1 & -C_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{T(\text{平移})}}_{V(\text{视图变换})} \begin{pmatrix} P_X \\ P_Y \\ P_Z \\ 1 \end{pmatrix}$$

视觉空间中的点 P_C　　世界空间中的点 P_W

图 3.13 推导视图矩阵

通常，$\boldsymbol{V}$ 矩阵与模型矩阵 $\boldsymbol{M}$ 合并成为一个模型-视图（model-view，MV）矩阵：

$$\text{MV} = \boldsymbol{V} * \boldsymbol{M}$$

之后，点 P_M 在从自己的模型空间通过如下一个步骤就可以直接转换至相机空间：

$$P_C = \mathrm{MV} * P_M$$

在复杂场景中，当我们需要对每个顶点，而非只是一个点做这个变换的时候，这种方法的好处就很明显了。通过预先计算 MV，对于空间中每个点的变换只需要我们进行一次矩阵乘法计算。之后，我们将会看到，我们可以将这个过程延伸到与计算更多的合并矩阵，以大量减少每个顶点的计算量。

3.8 投影矩阵

当我们设置好相机之后，就可以学习投影矩阵了。我们需要学习的两个重要的投影矩阵是：透视投影和正射投影。

3.8.1 透视投影矩阵

透视投影通过使用透视概念模仿我们看真实世界的方式，尝试让 2D 图像看起来像是 3D 的。物体近大远小，3D 空间中有的平行线用透视法画出来就不再平行。

透视法是 15 世纪初～16 世纪初文艺复兴时期的伟大发现之一，当时的画家开始绘制比前人更加真实的画作。

图 3.14 中是一个绝好的例子。卡洛·克里韦利在 1486 年绘制的《圣母领报》（又名《给



图 3.14 《圣母领报》（卡洛·克里韦利，1486 年）

圣·埃米迪乌斯报喜》目前收藏于伦敦国家美术馆[CR86]）。这幅画明显强烈地使用了透视——右侧建筑的左墙向后的线条戏剧性地一起倾斜。这种画法让人产生了深度感知和画中有3D空间的错觉。这个过程中，在现实中平行的线在画中并不平行。同样，在前景中的人物比在背景中的人物要大。虽然今天我们将这些视为理所当然的，不过算出实现它的变换矩阵还是需要一些数学分析的。

我们通过使用变换矩阵将平行线变为恰当的不平行线来实现这个效果。这个矩阵叫作**透视矩阵**或者**透视变换**，通过定义4个参数来进行视体（view volume）的构建。其中4个参数是纵横比、视场、投影平面或近剪裁平面、远剪裁平面。

只有在**远近剪裁平面**间的物体才会被渲染。近剪裁平面同时也是物体所投影到的平面，通常放在离眼睛或相机较近的位置（如图3.15左侧所示）。我们会在第4章中讨论如何为远剪裁平面选择合适的值。**视场**是可视空间的纵向角度。**纵横比**是远近剪裁平面的宽度比高度。通过这些元素所形成的形状叫作**视锥**（frustum），如图3.15所示。

透视矩阵用于将3D空间中的点变换至近剪裁平面上合适的位置，它的构建需要先计算q、A、B、C的值，之后用这些值来构建透视矩阵，如图3.16所示（推导过程见参考资料[FV95]）。

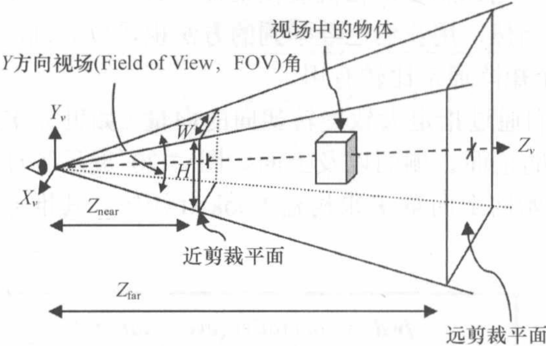


图3.15 透视视体或视锥

$$q = \frac{1}{\tan(\frac{\text{fieldOfView}}{2})}$$

$$A = \frac{q}{\text{aspectRatio}}$$

$$B = \frac{Z_{near} + Z_{far}}{Z_{near} - Z_{far}}$$

$$C = \frac{2 * (Z_{near} * Z_{far})}{Z_{near} - Z_{far}}$$

$$\begin{bmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

图3.16 构建透视矩阵

生成透视变换矩阵很容易，只需要将所描述的公式插入一个4×4矩阵。GLM库也包含了一个用于构建透视矩阵的函数glm::perspective()。

3.8.2 正射投影矩阵

在**正射投影**中，平行线仍然是平行的，即不使用透视，如图3.17所示。正射与透视相反，在视体中的物体不因其距相机距离做任何调整，而直接进行投影。

正射投影是一种平行投影，其中所有的投影都与投影平面垂直。正射矩阵通过如下参数构建：（a）从相机到投影平面的距离Z_{near}；（b）从相机到远剪裁平面的距离Z_{far}；（c）L、R、

T、和 B 的值，其中 L 和 R 分别是投影平面左右边界的 X 坐标，T 和 B 分别是投影平面上下边界的 Y 坐标，如图 3.18 所示。

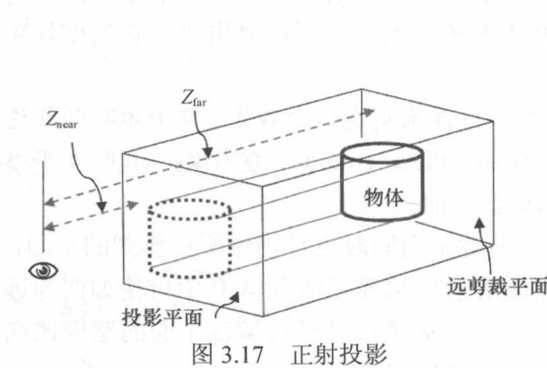


图 3.17 正射投影

$$\begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{1}{Z_{far}-Z_{near}} & -\frac{Z_{near}}{Z_{far}-Z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图 3.18 正射投影矩阵

并非所有平行投影都是正射投影，但是其他平行投影不在本书范围内。

平行投影与我们眼睛所见到的真实世界不同。但是它们在很多情况下都有其用处，比如投射阴影、进行 3D 剪裁以及 CAD（计算机辅助设计）中——用在 CAD 中是因为无论物体如何摆放，其尺寸都不变。无论如何，本书中绝大多数例子使用透视投影。

3.9 LookAt 矩阵

我们最后要学习的变换是 LookAt 矩阵。当你想要把相机放在某处并看向一个特定的位置时，就需要用到它了，如图 3.19 所示。当然，用我们已经学到的方法也可以做到，但是这个操作非常频繁，因此为它专门构建一个矩阵通常比较有用。

LookAt 变换依然由相机旋转决定。我们通过指定大致旋转朝向的向量（如世界 Y 轴）。通常，可以通过一系列叉积获得相机旋转的正面、侧面以及上面。图 3.20 展示了计算过程，从相机位置（眼睛）、目标位置以及初始向上向量 $\boldsymbol{Y}$ 来构建 LookAt 矩阵，其推导过程在参考资料[FV95]中。

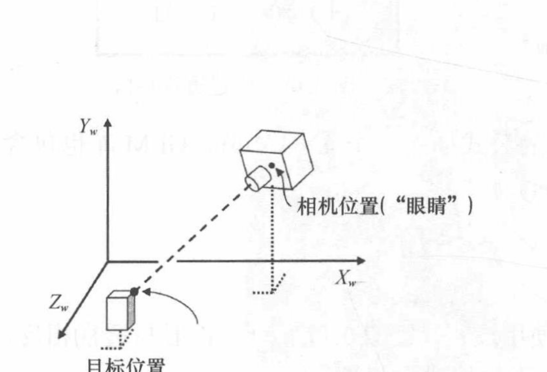


图 3.19 LookAt 的元素

$$\boldsymbol{fwd} = normalize(\boldsymbol{eye} - \boldsymbol{target})$$
$$\boldsymbol{side} = normalize(-\boldsymbol{fwd} \times \boldsymbol{Y})$$
$$\boldsymbol{up} = normalize(\boldsymbol{side} \times (-\boldsymbol{fwd}))$$

LookAt矩阵等于：

$$\begin{bmatrix} side_X & side_Y & side_Z & -(side \bullet eye) \\ up_X & up_Y & up_Z & -(up \bullet eye) \\ -fwd_X & -fwd_Y & -fwd_Z & -(-fwd \bullet eye) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图 3.20 LookAt 矩阵

我们可以将这个过程构建为一个 C++/OpenGL 实用函数，通过指定相机位置、目标位置

以及初始“向上”向量 ***Y***，构建一个 LookAt 矩阵。由于 GLM 中已经有一个用来构建 LookAt 矩阵的函数 glm::lookAt()，我们用它就可以了。稍后本书第 8 章生成阴影的时候会用到这个函数。

3.10 用来构建矩阵变换的 GLSL 函数

虽然 GLM 包含了许多预定义的 3D 变换函数，本章也已经涵盖了其中如平移、旋转和缩放，但 GLSL 只包含了基础的矩阵运算，如加法、合并等。因此，有时我们需要自己为 GLSL 写一些实用函数来构建 3D 变换矩阵，以在着色器中进行特定 3D 运算。用于存储这些矩阵的 GLSL 数据类型是 mat4。

GLSL 中用于初始化 mat4 矩阵的语法以列为单位读入值。前 4 个参数会放入第一列，接下来 4 个参数放入下一列，直到第四列，如下所示：

```
mat4 translationMatrix =
   mat4(1.0, 0.0, 0.0, 0.0, // 注意，这是最左列，而非第一行
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        tx, ty, tz, 1.0 );
```

构建如图 3.3 所示的平移矩阵。

程序 3.1 中包含了 5 个用于构建 4×4 平移、旋转和缩放矩阵的 GLSL 函数，每个函数对应于本章之前给出的一个公式。我们稍后在书中将会用到这些函数。

程序 3.1　在 GLSL 中构建变换矩阵

```
// 构建并返回平移矩阵
mat4 buildTranslate(float x, float y, float z)
{ mat4 trans = mat4(1.0, 0.0, 0.0, 0.0,
                    0.0, 1.0, 0.0, 0.0,
                    0.0, 0.0, 1.0, 0.0,
                    x, y, z, 1.0 );
  return trans;
}

// 构建并返回绕 X 轴的旋转矩阵
mat4 buildRotateX(float rad)
{ mat4 xrot = mat4(1.0, 0.0, 0.0, 0.0,
                   0.0, cos(rad), -sin(rad), 0.0,
                   0.0, sin(rad), cos(rad), 0.0,
                   0.0, 0.0, 0.0, 1.0 );
  return xrot;
}

// 构建并返回绕 Y 轴的旋转矩阵
mat4 buildRotateY(float rad)
{ mat4 yrot = mat4(cos(rad), 0.0, sin(rad), 0.0,
                   0.0, 1.0, 0.0, 0.0,
                   -sin(rad), 0.0, cos(rad), 0.0,
                   0.0, 0.0, 0.0, 1.0 );
  return yrot;
}
```

```
// 构建并返回绕 Z 轴的旋转矩阵
mat4 buildRotateZ(float rad)
{ mat4 zrot = mat4(cos(rad), -sin(rad), 0.0, 0.0,
                   sin(rad), cos(rad), 0.0, 0.0,
                   0.0, 0.0, 1.0, 0.0,
                   0.0, 0.0, 0.0, 1.0 );
  return zrot;
}

// 构建并返回缩放矩阵
mat4 buildScale(float x, float y, float z)
{ mat4 scale = mat4(x, 0.0, 0.0, 0.0,
                    0.0, y, 0.0, 0.0,
                    0.0, 0.0, z, 0.0,
                    0.0, 0.0, 0.0, 1.0 );
  return scale;
}
```

补充说明

在本章中我们看到了使用矩阵对点进行变换的例子。稍后，我们会将同样的变换应用于向量。要对向量 $\boldsymbol{V}$ 使用变换矩阵 $\boldsymbol{M}$ 进行与点相同的变换，一般需要计算 $\boldsymbol{M}$ 的逆转置矩阵，记为$(\boldsymbol{M}^{-1})^{\mathrm{T}}$，并用所得矩阵乘以 $\boldsymbol{V}$。在某些情况下，$\boldsymbol{M}=(\boldsymbol{M}^{-1})^{\mathrm{T}}$，在这些情况下只要用 $\boldsymbol{M}$ 就可以了。例如，本章中我们所见到的基础旋转矩阵与它们的逆转置矩阵相等，我们可以直接将他们应用于向量（同样也可以应用于点）。因此本书中有时候使用$(\boldsymbol{M}^{-1})^{\mathrm{T}}$对向量进行变换，有时候仅使用 $\boldsymbol{M}$。

本章中仍未讨论的一个技术是在空间中平滑地移动相机。这是一种很有用的技术，在制作游戏和 CGI 电影时更加明显，同时也适用于可视化、虚拟现实和 3D 建模。

我们也没有讲解所有给出的矩阵变换的推导过程（见其他资源，如参考资料[FV95]）。相反，我们努力做到简明地总结了基础 3D 图形学变种中必备的点、向量和矩阵运算。随着本书的推进，我们将会看到本章中方法的许多实际应用。

习题

3.1 修改程序 2.5，为顶点着色器添加程序 3.1 中的一个 buildRotate()函数，并将其应用到组成三角形的点上。其结果应该导致三角形从原来的方向进行旋转。这个旋转过程无须动画化。

3.2 （研究）在 3.4 节末尾，我们讲到欧拉角在某些情况下会导致瑕疵。其中最常见的叫作“万向节死锁”。描述万向节死锁，给出一个例子，并解释为什么万向节死锁会是个问题。

3.3 （研究）避免这些瑕疵的一种方法是使用四元数而非欧拉角。我们在本书中并没有学习四元数，但是 GLM 有一些四元数相关的类和函数。请独立学习四元数，并熟悉 GLM 中的四元数功能。

参考资料

[CR86] C. Crivelli, *The Annunciation, with Saint Emidius*, (National Gallery, London, England, 1486), accessed October 2018.

[EU76] L. Euler, "Formulae generals pro translatione quacunque coporum rigidorum" (General formulas for the translation of arbitrary rigid bodies), (Novi Commentarii academiae scientiarum Petropolitanae 20, 1776).

[FV95] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics – Principles and Practice*, 2nd ed. (Addison-Wesley, 1995).

[KU98] J. B. Kuipers, *Quaternions and Rotation Sequences* (Princeton University Press, 1998).

第 4 章　管理 3D 图形数据

使用 OpenGL 渲染 3D 图形通常需要将若干数据集发送给 OpenGL 着色器管线。举个例子，想要绘制一个简单的 3D 对象，比如一个立方体，你至少需要发送以下项目。

- 立方体模型的顶点。
- 控制立方体在 3D 空间中朝向表现的变换矩阵。

把数据发送给 OpenGL 管线还要更加复杂一点，有两种方式。

- 通过顶点属性的缓冲区。
- 直接发送给统一变量。

理解这两种机制具体如何工作非常重要，这样我们才能为每个要发送的项目选取合适的方法。

让我们从渲染一个简单的立方体开始。

4.1　缓冲区和顶点属性

想要绘制一个对象，它的顶点数据需要被发送给顶点着色器。通常会把顶点数据在 C++ 端放入一个缓冲区，并把这个缓冲区和着色器中声明的顶点属性相关联。要完成这件事，有好几个步骤，有些步骤只需要做一次，而如果是动画场景的话，有些步骤需要每帧都做一次：

只做一次的步骤——一般是在 init()中。

（1）创建一个缓冲区。

（2）将顶点数据复制到缓冲区。

每帧都要做的步骤——一般是在 display()中。

（1）启用包含了顶点数据的缓冲区。

（2）将这个缓冲区和一个顶点属性相关联。

（3）启用这个顶点属性。

（4）使用 glDrawArrays(...)绘制对象。

所有缓冲区通常在程序开始的时候统一创建，可以在 init()中，或者在被 init()调用的函数中。在 OpenGL 中，缓冲区被包含在*顶点缓冲对象*（Vertex Buffer Object，VBO）中，VBO 在 C++/OpenGL 应用程序中被声明和实例化。一个场景可能需要很多 VBO，所以常常会在 init()中生成并填充若干个 VBO，这样在你的程序需要绘制一个或多个 VBO 的时候就可以直接使用。

缓冲区使用特定的方式和顶点属性交互。当 glDrawArrays()被执行时，缓冲区中的数据开始流动，从缓冲区的开头开始，按顺序流过顶点着色器。像第 2 章中介绍的一样，顶点

着色器对每个顶点执行一次。3D 空间中的顶点需要 3 个数值，所以着色器中的顶点属性常常会以 vec3 类型接收到这 3 个数值。然后，对缓冲区中的每组这 3 个数值，着色器会被调用，如图 4.1 所示。

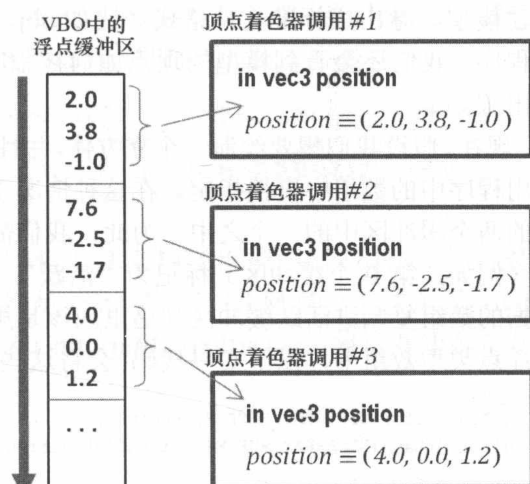


图 4.1　在 VBO 和顶点属性之间的数据传递

OpenGL 中还有一种相关的结构，叫作顶点数组对象（Vertex Array Object，VAO）。OpenGL 的 3.0 版本引入了 VAO，作为一种组织缓冲区的方法，让缓冲区在复杂场景中更容易操控。OpenGL 要求至少创建一个 VAO，对我们现在来说一个就够了。

举个例子，假设我们想要显示两个对象。在 C++端，我们可以声明一个 VAO 和两个相关的 VBO（每个对象一个），就像这样：

```
GLuint vao[1];      // OpenGL 要求这些数值以数组的形式指定
GLuint vbo[2];
…
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);
glGenBuffers(2, vbo);
```

glGenVertexArrays()和 glGenBuffers()这两个 OpenGL 命令分别创建 VAO 和 VBO，并返回它们的整数型 ID。我们把这些 ID 存进整数型数组 vao 和 vbo 中。这两个命令各自有两个参数，第一个是要创建多少个 ID，第二个是用来保存返回的 ID 的数组。glBindVertexArrays()命令的目的是将指定的 VAO 标记为“活跃”，这样生成的缓冲区①就会和这个 VAO 相关联。

每个缓冲区需要有在顶点着色器中声明的相应的顶点属性变量。顶点属性通常是着色器中首先被声明的变量。在我们的立方体例子中，用来接收立方体顶点的顶点属性可以在顶点着色器中这样声明：

```
layout (location = 0) in vec3 position;
```

关键字 in 意思是“输入”（input），表示这个顶点属性将会从缓冲区中接收数值（我们以后会看到，顶点属性也可以用来“输出”）。像我们之前看到的一样，“vec3”的意思是着色器的每次调用会抓到 3 个浮点类型数值（分别表示 x、y、z，它们组成一个顶点数据）。变量的名字是“position”。命令中“layout (location=0)”这部分叫作“layout 修饰符”，也就是我们把顶点属性和特定缓冲区关联起来的方法。这意味着，这个顶点属性的识别号是 0，我们后面会用到。

我们把一个模型的顶点加载到缓冲区（VBO）的方式取决于模型的顶点数值存储在哪里。在第 6 章中，我们将会看到通常如何使用建模工具（比如 Blender [BL16]或者 Maya [MA16]）

① 在这个例子中，我们声明了两个缓冲区，以强调我们常常会用到多个缓冲区。后面我们会用到额外的缓冲区来存储顶点相关的其他信息，比如颜色。现在，我们只用到了一个声明的缓冲区，所以如果只声明一个 VBO 也是足够的。

创建模型、导出成标准文件格式（比如.obj，在第6章会介绍）并导入到C++/OpenGL应用程序。我们还会看到模型的顶点如何被临时计算出来，或者在管线中使用细分着色器生成出来。

现在，假设我们想要绘制一个立方体，并且假定我们的立方体的顶点数据在C++/OpenGL应用程序中的数组中直接指定。在这种情况下，我们需要（a）将这些值复制到我们之前生成的两个缓冲区中的一个之中。为此，我们需要使用OpenGL的glBindBuffer()命令将缓冲区（例如，第0个缓冲区）标记为“活跃”；并且（b）使用glBufferData()命令将包含顶点数据的数组复制进活跃缓冲区（这里应该是第0个VBO）。假设顶点存储在名为vPositions的浮点类型数组中，以下C++代码[①]会将这些值复制到第0个VBO中：

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vPositions), vPositions, GL_STATIC_DRAW);
```

接下来，我们向display()中添加代码，将缓冲区中的值发送到着色器中的顶点属性。我们通过以下3个步骤来实现：（a）使用glBindBuffer()命令标记这个缓冲区为“活跃”，正如上所述；（b）将活跃缓冲区与着色器中的顶点属性相关联；（c）启用顶点属性。以下代码行实现了这些步骤：

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);                          // 标记第0个缓冲区为"活跃"
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);          // 将第0个属性关联到缓冲区
glEnableVertexAttribArray(0);                                   // 启用第0个顶点属性
```

现在，当我们执行glDrawArrays()时，第0个VBO中的数据将被传输给拥有位置0的layout修饰符的顶点属性中。这会将立方体的顶点数据发送到着色器。

4.2 统一变量

要想渲染一个场景以使它看起来是3D的，需要构建适当的变换矩阵，例如第3章中描述的那些，并将它们应用于模型的每个顶点。在顶点着色器中应用所需的矩阵运算是最有效的，并且习惯上会将这些矩阵从C++/OpenGL应用程序发送给着色器中的统一变量。

使用“uniform”关键字在着色器中声明统一变量。以下示例声明了用于存储模型-视图和投影矩阵的变量，足够我们的立方体程序使用：

```
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

关键字“mat4”表示这些是4×4矩阵。这里我们将用来保存模型-视图矩阵的变量命名为mv_matrix，并将用来保存投影矩阵的变量命名为proj_matrix。因为3D变换是4×4的，因此mat4是GLSL着色器统一中常用的数据类型。

将数据从C++/OpenGL应用程序发送到统一变量需要执行以下步骤：（a）获取统一变量的引用；（b）将指向所需数值的指针与获取的统一引用相关联。在我们的立方体例子中，

① 请注意，这里，我们第一次避免描述一个或多个OpenGL调用中的每一个参数。如第2章所述，我们建议读者根据需要利用OpenGL文档来获取此类详细信息。

假设链接的渲染程序保存在名为“renderingProgram”的变量中，以下代码行表示，我们要把模型-视图和投影矩阵发送到两个统一变量 mv_matrix 和 proj_matrix 中去：

```
mvLoc = glGetUniformLocation(renderingProgram,"mv_matrix");      // 获取着色器程序中统一变量的位置
projLoc = glGetUniformLocation(renderingProgram,"proj_matrix");
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));   // 将矩阵数据发送到统一变量中
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
```

在上面的例子中，我们假设已经利用 GLM 工具来构建模型-视图和投影矩阵变换 mvMat 和 pMat，稍后将会更详细地讨论。它们是 mat4 类型（GLM 的一个类）的。GLM 函数调用 value_ptr()返回对矩阵数据的引用，glUniformMatrix4fv()需要将这些矩阵值传递给统一变量。

4.3 顶点属性插值

相较于如何处理统一变量，了解如何在 OpenGL 管线中处理顶点属性非常重要。回想一下，在片段着色器光栅化之前，由顶点定义的图元（例如，三角形）被转换为片段。光栅化过程会**线性插值顶点属性**值，以便显示的像素能无缝连接建模的曲面。

相比之下，统一变量的行为类似于初始化过的常量，并且在每次顶点着色器调用（即从缓冲区发送的每个顶点）中保持不变。统一变量本身不是插值的；无论有多少顶点，它始终包含相同的值。

光栅着色器对顶点属性进行的插值在很多方面都很有用。稍后，我们将使用光栅化来插值颜色、纹理坐标和曲面法向量。重要的是要理解通过缓冲区发送到顶点属性的所有值都将在管线中被进一步插值。

我们在顶点着色器中看到顶点属性被声明为“in”，表示它们从缓冲区接收值。顶点属性还可以改为声明为“out”，这意味着它们会将值发送到管线中的下一个阶段。例如，顶点着色器中的以下声明将指定一个名为“color”的顶点属性，该属性输出 vec4 类型的值：

```
out vec4 color;
```

没有必要为顶点位置声明一个“out”变量，因为 OpenGL 有一个内置的 vec4 变量用于此目的，它的名字叫作 gl_Position。在顶点着色器中，我们将矩阵变换应用于传入的顶点（之前声明为位置的顶点），并将结果赋值给 gl_Position：

```
gl_Position = proj_matrix * mv_matrix * position;
```

然后，变换后的顶点将自动输出到光栅着色器，最终将相应的像素位置发送到片段着色器。

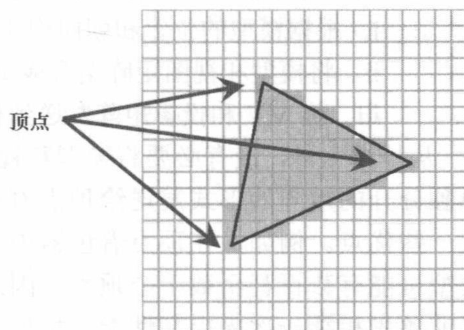


图 4.2 顶点的光栅化

光栅化过程如图 4.2 所示。在 glDrawArrays()函数中指定 GL_TRIANGLES 时，光栅化是逐个

三角形完成的。首先沿着连接顶点的线开始插值，其精度级别和像素显示密度相关。然后通过沿连接边缘像素的水平线插值来填充三角形的内部空间中的像素。

4.4 模型-视图和透视矩阵

渲染3D对象的一个基础步骤是创建适当的变换矩阵并将它们发送到统一变量，就像我们在4.2节中所做的那样。我们首先定义3个矩阵：

- 一个模型矩阵；
- 一个视图矩阵；
- 一个透视矩阵。

模型矩阵在世界坐标空间中表示对象的位置和朝向。每个模型都有自己的模型矩阵，如果模型移动，则需要不断重建该矩阵。

视图矩阵移动并旋转世界中的模型，以模拟相机在所需位置的效果。回忆一下第2章，OpenGL相机存在于位置(0,0,0)并且面向负Z轴。为了模拟以某种方式移动的相机的表现，我们需要向相反的方向移动物体本身。例如，将摄像机向右移动会导致场景中的物体看起来像是向左移动；虽然OpenGL相机是固定的，但我们可以通过把对象向左移动的方式，让摄像机看起来向右移动了。

透视矩阵是一种变换，它根据所需的视锥提供3D效果，如前面第3章所述。

了解何时计算每种类型的矩阵也很重要。永远不会改变的矩阵可以在init()中构建，但那些会改变的矩阵需要在display()中构建，以便为每个帧重建它们。我们假设模型是动画的，相机是可移动的，那么：

- 需要为每个模型和每个帧都创建模型矩阵；
- 视图矩阵需要每帧创建一次（因为相机可以移动），但是对于在这一帧期间渲染的所有对象，它都是一样的；
- 透视矩阵只需要创建一次［在init()中］，它需要使用屏幕窗口的宽度和高度（以及所需的视锥体参数），除非调整窗口大小，否则它通常保持不变。

然后在display()函数中生成模型和视图转换矩阵，如下所示。

（1）根据所需的摄像机位置和朝向构建视图矩阵。

（2）对于每个模型，进行以下操作。

  i. 根据模型的位置和朝向构建模型矩阵。

  ii. 将模型和视图矩阵结合成单个“MV”矩阵。

  iii. 将MV和投影矩阵发送到相应的着色器统一变量。

从技术上讲，没有必要将模型和视图矩阵合并成一个矩阵。也就是说，它们也可以用单独分开的矩阵的形式发送给顶点着色器。然而，将它们合并，并保持透视矩阵分离，有一些优点。例如，在顶点着色器中，模型中的每个顶点都需要乘以矩阵。由于复杂的模型可能有数百甚至数千个顶点，因此可以通过在将模型和视图矩阵发送到顶点着色器之前预先相乘一次来提高性能。稍后，我们将看到为什么需要将透视矩阵分开以用于光照的目的。

4.5 我们的第一个 3D 程序——一个 3D 立方体

是时候将所有部分组合在一起了！为了构建一个完整的 C++/OpenGL/GLSL 系统并在 3D“世界”中渲染我们的立方体，到目前为止介绍过的所有机制都需要被整合在一起，并完美协调。我们可以重用我们之前在第 2 章中看到的一些代码。具体来说，我们不会再重复讲解以下这些用来读取包含着色器代码的文件，编译和链接它们，以及检测 GLSL 错误的函数；事实上，回想一下，我们已将它们移到“Utils.cpp”文件中：

- `createShaderProgram()`
- `readShaderSource()`
- `checkOpenGLError()`
- `printProgramLog()`
- `printShaderLog()`

在给定了 *Y* 轴的指定视场角、屏幕纵横比以及所需的近、远剪裁平面（在 4.9 节中讨论了如何为近剪裁平面和远剪裁平面选择适当的值）的情况下，我们还需要一个构建透视矩阵的工具函数。虽然我们可以自己轻松编写这样的函数，但 GLM 已经包含了一个：

```
glm::perspective(<field of view>, <aspect ratio>, <near plane>, <far plane>);
```

我们现在可以构建完整的 3D 立方体程序了，如下面的程序 4.1 所示。

程序 4.1 简单的红色立方体

```
C++/OpenGL 应用程序

#include <GL\glew.h>
#include <GLFW\glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm\glm.hpp>
#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>
#include "Utils.h"
using namespace std;

#define numVAOs 1
#define numVBOs 2

float cameraX, cameraY, cameraZ;
float cubeLocX, cubeLocY, cubeLocZ;
GLuint renderingProgram;
GLuint vao[numVAOs];
GLuint vbo[numVBOs];

// 分配在 display() 函数中使用的变量空间，这样它们就不必在渲染过程中分配
GLuint mvLoc, projLoc;
int width, height;
float aspect;
glm::mat4 pMat, vMat, mMat, mvMat;
```

```
void setupVertices(void) {    // 36个顶点，12个三角形，组成了放置在原点处的2×2×2立方体
   float vertexPositions[108] = {
      -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
      1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f,
      1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f,
      1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f,
      1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
      -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
      -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f,
      -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f,
      -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f,
      1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f,
      -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
      1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, -1.0f
   };
   glGenVertexArrays(1, vao);
   glBindVertexArray(vao[0]);
   glGenBuffers(numVBOs, vbo);

   glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
   glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
}

void init(GLFWwindow* window) {
   renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");
   cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
   cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f; // 沿Y轴下移以展示透视
   setupVertices();
}

void display(GLFWwindow* window, double currentTime) {
   glClear(GL_DEPTH_BUFFER_BIT);
   glUseProgram(renderingProgram);

   // 获取MV矩阵和投影矩阵的统一变量
   mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
   projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");

   // 构建透视矩阵
   glfwGetFramebufferSize(window, &width, &height);
   aspect = (float)width / (float)height;
   pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f); // 1.0472 radians = 60 degrees

   // 构建视图矩阵、模型矩阵和视图-模型矩阵
   vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
   mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cubeLocX, cubeLocY, cubeLocZ));
   mvMat = vMat * mMat;

   // 将透视矩阵和MV矩阵复制给相应的统一变量
   glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
   glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));

   // 将VBO关联给顶点着色器中相应的顶点属性
   glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
   glEnableVertexAttribArray(0);

   // 调整OpenGL设置，绘制模型
   glEnable(GL_DEPTH_TEST);
```

```
    glDepthFunc(GL_LEQUAL);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}

int main(void) {                                    // main()和之前的没有变化
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter 4 - program 1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {
        display(window, glfwGetTime());
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

顶点着色器（文件名："vertShader.glsl"）

```
#version 430

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{ gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

片段着色器（文件名："fragShader.glsl"）

```
#version 430

out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{ color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

程序 4.1 的输出如图 4.3 所示（见彩插）。让我们仔细看看程序 4.1 中的代码。重要的是，我们要了解所有部分的工作原理以及它们如何协同工作。

下面查看由 init()调用的函数 setupVertices()。在此函数的开头，声明一个名为 vertexPositions 的数组，其中包含 36 个组成立方体的顶点。首先你可能想知道为什么这个立方体有 36 个顶点，逻辑上一个立方体应该只需要 8 个顶点。答案是我们需要用三角形来构建我们的立方体，因此 6 个立方体面中的每一个都需要由两个三角形构成，总共 6×2=12

个三角形（见图4.4）。由于每个三角形由3个顶点指定，因此总共有36个顶点。由于每个顶点具有3个值(x, y, z)，因此数组中总共有36×3=108个值。确实，每个顶点都参与了多个三角形的组成，但我们仍然分别指定每个顶点，因为现在我们会将每个三角形的顶点分别发送到管线。

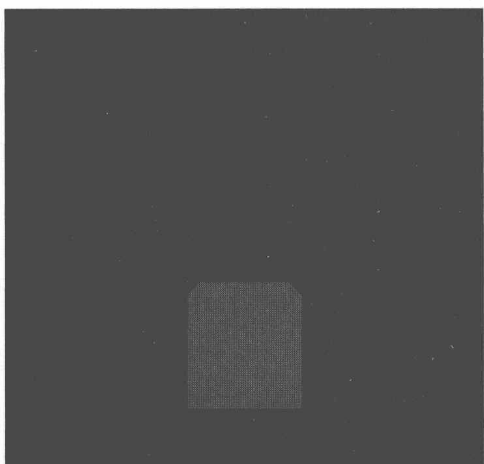


图4.3 程序4.1的输出。从(0,0,8)看位于(0,−2,0)的红色立方体

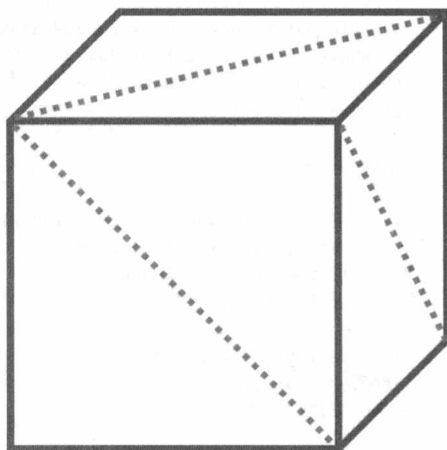


图4.4 由三角形组成的立方体

立方体在它自己的坐标系中定义，中心为(0,0,0)，它的角在 X、Y 和 Z 这3条轴上分别位于−1.0～+1.0。setupVertices()的其余部分建立了VAO和两个VBO（尽管只使用了一个）并将立方体顶点加载到第0个VBO缓冲区中。

请注意，init()函数负责执行只需要执行一次的任务：读取着色器代码并构建渲染程序，并将立方体顶点加载到VBO中[通过调用setupVertices()]。请注意，它还给定了立方体和相机在世界中的位置。稍后我们将为立方体设置动画，并了解如何移动相机，到那个时候我们可能需要去除这个固定的位置。

现在让我们看一下 display()函数。回想一下，display()可以被重复调用，并且调用它的速率被称为帧率。也就是说，通过不断地快速绘制和重绘场景或帧，就可以实现动画。通常需要在渲染帧之前清除深度缓冲区，以便正确地进行隐藏面消除（不清除深度缓冲区有时会导致每个曲面都被移除，从而导致完全黑屏）。默认情况下，OpenGL 中的深度值范围为 0.0～1.0。调用 glClear(GL_DEPTH_BUFFER_BIT)就可以清除深度缓冲区，这会使用默认值（通常为1.0）来填充深度缓冲区。

接下来，display()通过调用glUseProgram()来启用着色器，在GPU上安装GLSL代码。回想一下，这并不会运行着色器程序，但它会让后续的 OpenGL 调用能够确定着色器的顶点属性和统一变量位置。display()函数接下来获取统一变量位置，构建透视、视图和模型矩阵①，将视图和模型矩阵结合成单一的MV矩阵，并将透视和MV矩阵赋值给它们相应的统一变量。在这里，值得注意的是对translate()函数的GLM调用的形式：

① 精明的读者可能会注意到，并不需要每次调用 display()时都构建透视矩阵，因为它的值不会改变。这在一部分情况下是正确的——如果用户在程序运行时调整窗口大小，则需要重新计算透视矩阵。在4.11节中，我们将更有效地处理这种情况，并且在此过程中，我们会将透视矩阵的计算从display()移到init()函数。

```
vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
```

看起来有点神秘的调用通过以下方式构建了一个变换矩阵：从单位矩阵开始（使用 glm::mat4(1.0f)构造函数）和以向量的形式指定变换值（使用 glm::vec3(x,y,z)构造函数）。许多 GLM 变换操作使用这种方法。

接下来，display()函数启用了包含立方体顶点数据的缓冲区，并将其附加到第 0 个顶点属性，以准备将顶点数据发送到着色器。

display()函数做的最后一件事是通过调用 glDrawArrays()来绘制模型，指定模型由三角形组成并且总共有 36 个顶点。对 glDrawArrays()的调用通常在其他调整这个模型的渲染设置的命令之前。[①]在这个例子中，有两个这样的命令，这两个命令都与深度测试相关。回忆一下第 2 章，OpenGL 使用深度测试来进行隐藏面消除。在这里，我们启用深度测试并指定希望 OpenGL 使用的特定深度测试。此处显示的设置对应第 2 章中的说明；在本书的后续内容中，我们将看到这些命令的其他用途。

最后，说一说着色器。首先，请注意它们都包含相同的统一变量声明块。虽然并不总是一定要这样做，但在特定渲染程序中的所有着色器中包含相同的统一变量声明块通常是一种好习惯。

还要注意，顶点着色器中传入的顶点属性的 position 变量上是否存在 layout 修饰符。由于它的位置被指定为“0”，因此 display()函数可以简单地通过在 glVertexAttribPointer()函数调用的第一个参数和 glEnableVertexAttribArray()函数调用中使用 0 来引用此变量。请注意，position 顶点属性被声明为 vec3 类型，因此需要将其转换为 vec4 类型，以便与将要用它乘以的 4×4 矩阵兼容。这个转换是用 vec4(position,1.0)完成的，它用名为“position”的变量构建一个 vec4，在新添加的第四个点中放置一个值 1.0。

顶点着色器中的乘法将矩阵变换应用于顶点，将其转换为相机空间（请注意从右到左的结合顺序）。这些值被放入内置的 OpenGL 输出变量 gl_Position 中，然后继续通过管线并由光栅着色器进行插值。

然后插值后的像素位置（称为片段）被发送到片段着色器（Fragment Shader）。回想一下，片段着色器的主要目的是设置输出像素的颜色。以类似于顶点着色器的方式，片段着色器逐个处理像素，并为每个像素单独调用。在我们的例子中，它固定地输出对应于红色的值。由于前面指出的原因，统一变量已包含在片段着色器中，即使它们在此示例中并未被使用。

图 4.5 展示了从 C++/OpenGL 应用程序开始并通过管线的数据流概况。

让我们对着色器进行一些轻微的修改。特别是，我们将根据每个顶点的位置为每个顶点指定一种颜色，并将该颜色放在输出的顶点属性 varyingColor 中。同样，修改片段着色器以接收传入的颜色（由光栅着色器插值）并使用它来设置输出像素的颜色。请注意，代码中也将位置乘以 1/2，然后加 1/2，以将取值范围从[−1…+1]转换为[0…1]。还要注意的是，通常约定在程序员定义的插值顶点属性变量名称中包含单词“varying”。每个着色器中的更改都被高亮了，结果如下所示。

---

① 通常，这些调用可以放在 init()而不是 display()中。但是，在绘制具有不同属性的多个对象时，必须将其中一个或多个放在 display()中。为简单起见，我们总是将它们放在 display()中。

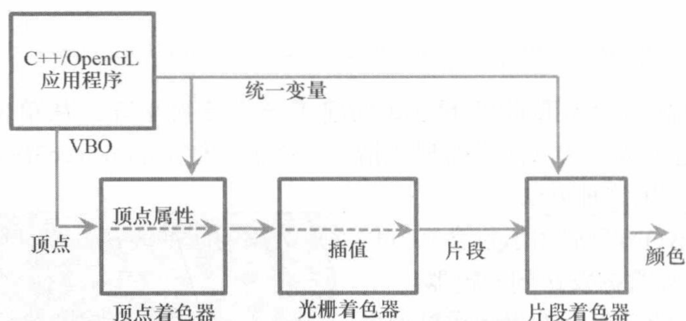


图 4.5 程序 4.1 的数据流

修改后的顶点着色器：

```
#version 430

layout (location=0) in vec3 position;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec4 varyingColor;

void main(void)
{ gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
   varyingColor = vec4(position,1.0) * 0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}
```

修改后的片段着色器：

```
#version 430

in vec4 varyingColor;

out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{ color = varyingColor;
}
```

请注意，因为颜色是从顶点着色器在顶点属性（varyingColor）中发出的，所以它们也由光栅着色器进行插值！它的效果可以在图 4.6（见彩插）中看到，从一个角到另一个角的颜色在整个立方体中明显是被插值了。

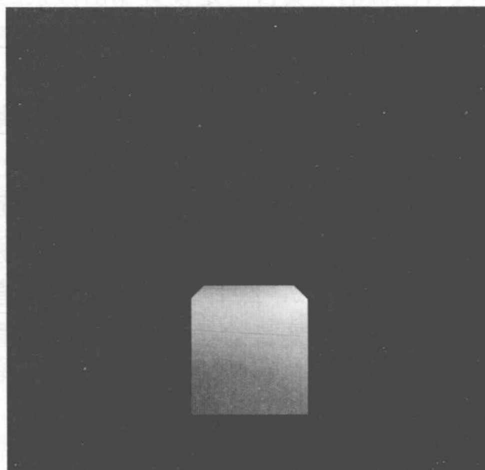


图 4.6 有插值颜色的立方体

另请注意，顶点着色器中的“out”变量 varyingColor 也是片段着色器中的“in”变量。两个着色器知道顶点着色器中的哪个变量提供片段着色器中的哪个变量，因为它们在两个着色器中具有相同的名称“varyingColor”。

由于我们的 main()函数包含一个渲染循

环，我们可以像在程序 2.6 中那样为我们的立方体设置动画，方法是使用基于时间变化的平移和旋转来构建模型矩阵。例如，程序 4.1 中 display()函数中的代码可以修改如下（突出显示更改）：

```
glClear(GL_DEPTH_BUFFER_BIT);
glClear(GL_COLOR_BUFFER_BIT);
. . .
// 使用当前时间来计算 x，y 和 z 的不同变换
tMat = glm::translate(glm::mat4(1.0f),
   glm::vec3(sin(0.35f*currentTime)*2.0f, cos(0.52f*currentTime)*2.0f, sin(0.7f*currentTime)*2.0f));
rMat = glm::rotate(glm::mat4(1.0f), 1.75f*(float)currentTime, glm::vec3(0.0f, 1.0f, 0.0f));
rMat = glm::rotate(rMat, 1.75f*(float)currentTime, glm::vec3(1.0f, 0.0f, 0.0f));
rMat = glm::rotate(rMat, 1.75f*(float)currentTime, glm::vec3(0.0f, 0.0f, 1.0f));
// 用 1.75 来调整旋转速度

mMat = tMat * rMat;
```

在模型矩阵中使用当前时间（以及各种三角函数）会使立方体看起来在空间中翻滚。请注意，添加此动画说明了每次通过 display()清除深度缓冲区以确保正确进行隐藏面消除的重要性。如图 4.6 所示，它还需要清除颜色缓冲区；否则，立方体会在移动时留下痕迹。

translate()和 rotate()函数是 GLM 库的一部分。另外，请注意最后一行中的矩阵乘法——操作中 tMat 和 rMat 的顺序很重要。它计算两个变换的结合，平移放在左边，旋转放在右边。当顶点随后乘以此矩阵时，计算从右到左进行，这意味着首先完成旋转，然后才是平移。变换的应用顺序很重要，改变顺序会导致不同的行为。图 4.7 显示了为立方体设置了动画后显示的一些帧。



图 4.7 为 3D 立方体设置动画（“翻滚”）

4.6 渲染一个对象的多个副本

现在将我们学到的知识扩展到渲染多个对象。在我们解决在单个场景中渲染多种不同的模型的常见情况之前，让我们先考虑更简单的情形——同一模型多次出现。例如，假设我们希望扩展前面的示例，以便呈现“一大群”（24 个）翻滚的立方体。我们可以将 display()函数中构建 MV 矩阵并绘制立方体的代码（如下所示突出显示部分），移动到一个执行 24 次的循环中来完成此操作。我们利用循环变量来计算立方体的旋转和平移参数，以便每次绘制立方体时，都会构建不同的模型矩阵。（我们还将摄像机放置在正 *Z* 轴的下方，这样我们就可以看到所有的立方体。）图 4.8 显示了一帧由此产生的动画场景。

```
void display(GLFWwindow* window, double currentTime) {
   . . .
   for (i=0; i<24; i++)
   { tf = currentTime + i;      // tf == "time factor（时间因子）", 声明为浮点类型
     tMat = glm::translate(glm::mat4(1.0f), glm::vec3(sin(.35f*tf)*8.0f, cos(.52f*tf)*8.0f,
                                                                        sin(.70f*tf)*8.0f));
     rMat = glm::rotate(glm::mat4(1.0f), 1.75f*tf, glm::vec3(0.0f, 1.0f, 0.0f));
     rMat = glm::rotate(rMat, 1.75f*tf, glm::vec3(1.0f, 0.0f, 0.0f));
     rMat = glm::rotate(rMat, 1.75f*tf, glm::vec3(0.0f, 0.0f, 1.0f));
     mMat = tMat * rMat;
     mvMat = vMat * mMat;

     glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
     glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));

     glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
     glEnableVertexAttribArray(0);

     glEnable(GL_DEPTH_TEST);
     glDepthFunc(GL_LEQUAL);
     glDrawArrays(GL_TRIANGLES, 0, 36);
   }
}
```

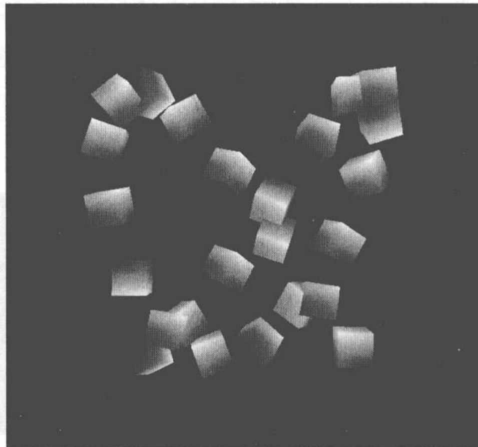


图 4.8　多个翻滚的立方体

实例化

实例化（Instancing）提供了一种机制，可以只用一个 C++/OpenGL 调用就告诉显卡渲染一个对象的多个副本。这可以带来显著的性能好处，特别是当有数千甚至数百万的对象被绘制时——例如渲染在场地中的许多花朵的时候。

我们首先将我们的 C++/OpenGL 应用程序中的 glDrawArrays()调用改为 glDrawArrays-Instanced()。这样，我们就可以要求 OpenGL 绘制尽可能多的副本。我们可以指定绘制如下 24 个立方体：

```
glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 24);
```

使用实例化时，顶点着色器可以访问内置变量 gl_InstanceID，这是一个整数，指向当前

正在处理对象的第几个实例。

为了使用实例化来重复我们以前的翻滚立方体示例，我们需要将构建不同模型矩阵的计算[先前在 display()中的循环内实现]移动到顶点着色器中。由于 GLSL 不提供平移或旋转函数，并且我们无法从着色器内部调用 GLM，因此我们需要使用程序 3.1 中的工具函数。我们还需要将 C++/OpenGL 应用程序中的“时间因子”通过统一变量传递给顶点着色器。我们还需要将模型和视图矩阵传递到单独的统一变量中，因为对每个立方体的模型矩阵都需要进行旋转计算。我们对代码的修改，包括 C++/OpenGL 应用程序中的修改以及新的顶点着色器中的修改，如程序 4.2 所示。

程序 4.2 实例化——24 个动画立方体

```
顶点着色器：

#version 430
layout (location=0) in vec3 position;

uniform mat4 m_matrix;                  // 这些是分开的模型和视图矩阵
uniform mat4 v_matrix;
uniform mat4 proj_matrix;
uniform float tf;                       // 用于动画和放置立方体的时间因子

out vec4 varyingColor;

mat4 buildRotateX(float rad);           // 矩阵变换工具函数的声明
mat4 buildRotateY(float rad);           // GLSL 要求函数先声明后调用
mat4 buildRotateZ(float rad);
mat4 buildTranslate(float x, float y, float z);

void main(void)
{ float i = gl_InstanceID + tf;         // 取值基于时间因子，但是对每个立方体示例也都是不同的
  float a = sin(2.0 * i) * 8.0;         // 这些是用来平移的 x、y、z 分量
  float b = sin(3.0 * i) * 8.0;
  float c = sin(4.0 * i) * 8.0;

  // 构建旋转和平移矩阵，将会应用于当前立方体的模型矩阵
  mat4 localRotX = buildRotateX(1000*i);
  mat4 localRotY = buildRotateY(1000*i);
  mat4 localRotZ = buildRotateZ(1000*i);
  mat4 localTrans = buildTranslate(a,b,c);

  // 构建模型矩阵，然后是模型-视图矩阵
  mat4 newM_matrix = m_matrix * localTrans * localRotX * localRotY * localRotZ;
  mat4 mv_matrix = v_matrix * newM_matrix;

  gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
  varyingColor = vec4(position,1.0) * 0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}

// 构建平移矩阵的工具函数（来自第 3 章）
mat4 buildTranslate(float x, float y, float z)
{ mat4 trans = mat4(1.0, 0.0, 0.0, 0.0,
                    0.0, 1.0, 0.0, 0.0,
                    0.0, 0.0, 1.0, 0.0,
                    x, y, z, 1.0 );
  return trans;
}
```

```
// 用来绕 X、Y、Z 轴旋转的类似函数（也来自第 3 章）
. . .

C++/OpenGL 应用程序（在 display()函数中）
  . . .
  // 构建（和变换）mMat 的计算被移动到顶点着色器中去了
  // 在 C++ 应用程序中不再需要构建 MV 矩阵
  glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(vMat));     // 着色器需要视图矩阵的统一变量
  timeFactor = ((float)currentTime);                                // 为了获得时间因子信息
  tfLoc = glGetUniformLocation(renderingProgram, "tf");             // （着色器也需要它）
  glUniform1f(tfLoc, (float)timeFactor);
  . . .
  glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 24);
```

程序 4.2 的输出结果与前一个示例相同，可以在前面的图 4.8 中看到。

实例化让我们可以极大地扩展对象的副本数量；在这个例子中，即使对于很普通的 GPU，实现 100 000 个立方体的动画仍然是可行的。对代码的更改主要是一些常量的修改，是为了将大量立方体进一步分散开，如下所示。

顶点着色器如下：

```
. . .
float a = sin(203.0 * i/8000.0) * 403.0;
float b = cos(301.0 * i/4001.0) * 401.0;
float c = sin(400.0 * i/6003.0) * 405.0;
. . .
```

C++/OpenGL 应用程序如下：

```
. . .
cameraZ = 420.0f; // 将摄像机沿着 Z 轴再移远一些，以看到更多的立方体
. . .
glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 100000);
```

输出结果如图 4.9 所示。

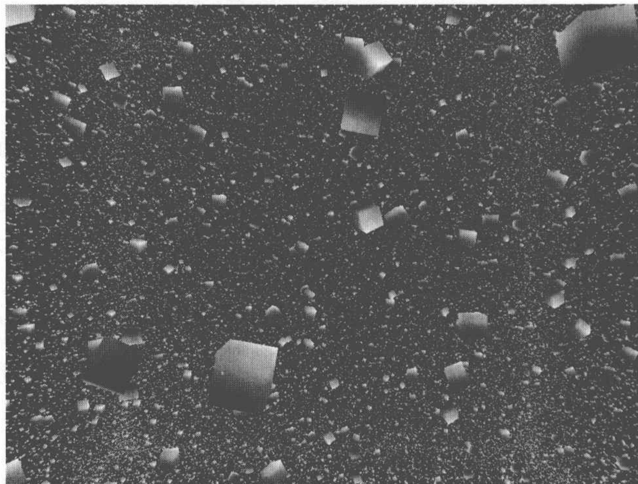


图 4.9 实例化：100 000 个动画立方体

4.7　在同一个场景中渲染多个不同模型

要在单个场景中渲染多个模型，一种简单的方法是为每个模型使用单独的缓冲区。每个模型都需要自己的模型矩阵，这样我们就需要为我们渲染的每个模型生成一个新的模型-视图矩阵。还需要为每个模型单独调用 glDrawArrays()。因此，我们需要修改 init()和 display()函数。

另一个考虑因素是我们是否需要为我们想要绘制的每个对象使用不同的着色器或不同的渲染程序。事实证明，在许多情况下，我们可以为我们绘制的各种对象使用相同的着色器（以及相同的渲染程序）。只有当它们由不同的图元（例如线而不是三角形）组成，或者涉及复杂的照明或其他效果的时候，我们才会需要为各种对象使用不同的渲染程序。目前并没有这么复杂，因此我们可以重用相同的顶点和片段着色器，而只需修改我们的 C++/OpenGL 应用程序，以便在调用 display()时将各个模型发送给管线。

让我们继续添加一个简单的金字塔，这样我们的场景就包括一个立方体和一个金字塔。程序 4.3 中显示了对代码的相关修改。我们突出显示了一些关键细节，例如当我们指定使用这个还是那个缓冲区的地方，以及我们指定模型中包含的顶点数的地方。注意，金字塔由 6 个三角形组成——侧面 4 个，底面 2 个，总共 6×3=18 个顶点。

包含立方体和金字塔的场景显示结果如图 4.10 所示。

程序 4.3　立方体和金字塔

```
void setupVertices() {
   float cubePositions[108] =
   { -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
      1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f,
         ... 立方体顶点的数据和以前一样
   };

   // 金字塔有 18 个顶点，由 6 个三角形组成（侧面 4 个，底面 2 个）
   float pyramidPositions[54] =
   { -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,      // 前面
     1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,      // 右面
     1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,    // 后面
     -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,    // 左面
     -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,   // 底面 — 左前一半
     1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f    // 底面 — 右后一半
   };
   glGenVertexArrays(numVAOs, vao); // 我们需要至少 1 个 VAO
   glBindVertexArray(vao[0]);
   glGenBuffers(numVBOs, vbo); // 我们需要至少 2 个 VBO

   glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
   glBufferData(GL_ARRAY_BUFFER, sizeof(cubePositions), cubePositions, GL_STATIC_DRAW);

   glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
   glBufferData(GL_ARRAY_BUFFER, sizeof(pyramidPositions), pyramidPositions, GL_STATIC_DRAW);
}

void display(GLFWwindow* window, double currentTime) {
   . . .
   // 像之前一样清除颜色和深度缓冲区（此处省略）
```

```
    // 像之前一样使用渲染程序并获取统一变量位置（此处省略）
    // 像之前一样计算投影矩阵（此处省略）
    . . .

    // 只计算一次视图矩阵，用于两个对象

    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));

    // 绘制立方体（使用 0 号缓冲区）

    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cubeLocX, cubeLocY, cubeLocZ));
    mvMat = vMat * mMat;

    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));

    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    // 绘制金字塔（使用 1 号缓冲区）

    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(pyrLocX, pyrLocY, pyrLocZ));
    mvMat = vMat * mMat;

    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));

    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glDrawArrays(GL_TRIANGLES, 0, 18);
}
```

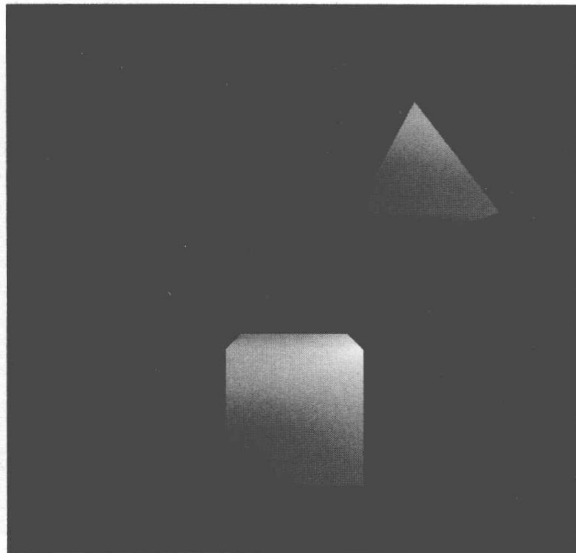


图 4.10 3D 立方体和金字塔

关于程序 4.3 的其他一些值得注意的小细节如下所示。

- 需要声明变量 pyrLocX、pyrLocY 和 pyrLocZ，然后在 init()中将它们初始化为所需的金字塔的位置，就像对立方体位置所做的那样。
- 在 display()的开始构建视图矩阵 vMat，然后在立方体和金字塔的模型-视图矩阵中都用到。
- 顶点和片段着色器代码被省略了——它们和 4.5 节中的一样。

4.8 矩阵堆栈

到目前为止，我们渲染的模型都是由一组顶点构成的。然而，实际上我们通常希望通过组装较小的简单模型来构建复杂的模型。例如，可以通过分别绘制头部、身体、腿部和手臂来创建“机器人”的模型，这当中每个部件都是一个单独的模型。以这种方式构建的对象通常称为*分层模型*。构建分层模型的棘手部分是跟踪所有模型-视图矩阵并确保它们完美协调——否则机器人可能会散成几块！

分层模型不仅可用于构建复杂对象——它们还可以用来生成复杂场景。例如，考虑一下我们的行星地球围绕太阳旋转的方式，以及月球围绕地球旋转的方式。这样的一个场景如图 4.11①所示。计算月球在太空中的实际路径可能很复杂。然而，如果我们能够组合代表两条简单圆形路径的变换——月球围绕地球旋转的路径和地球围绕太阳旋转的路径——我们就能避免直接计算月球的轨迹。

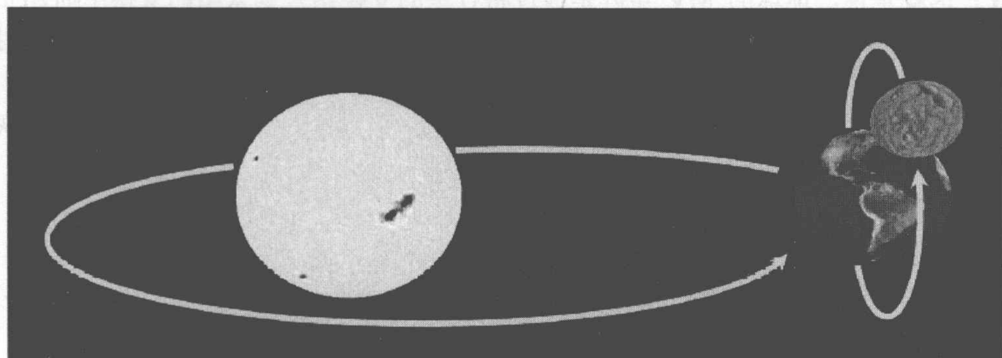


图 4.11 行星系统动画（太阳和地球的纹理来自[HT16]，月球纹理来自[NA16]）

事实证明，我们可以使用*矩阵堆栈*轻松地完成此操作。顾名思义，矩阵堆栈是一堆变换矩阵。正如我们将看到的，矩阵堆栈使得创建和管理复杂的分层对象和场景变得容易，它使得变换可以构建在其他变换之上（或者从其他变换中被移除）。

OpenGL 有一个内置的矩阵堆栈，但作为旧的固定功能（非可编程）管线的一部分，它早已被弃用[OL16]。但是，C++标准模板库（STL）有一个名为“stack”的类，通过使用它构建 mat4 的堆栈，它可以相对简单直接地当作矩阵堆栈使用。正如我们将看到的，复杂场景

---

① 是的，我们知道月球并不沿着这种“垂直”轨道绕地球旋转，而是围绕地球绕太阳旋转的共面。我们选择这个轨道使我们的程序执行更容易理解。

中通常需要的许多模型、视图和模型-视图矩阵可以由单个 stack<glm::mat4>实例替换。

我们将首先检查实例化和使用 C++堆栈的基本命令，然后使用一个堆栈来构建复杂的动画场景。我们将通过以下方法使用 C++堆栈类。

- push()：在堆栈顶部创建一个新的条目。我们通常会把目前在堆栈顶部的矩阵复制一份，并和其他的变换结合，然后再利用这个命令把新的矩阵副本推入堆栈中。
- pop()：移除（并返回）最顶部的矩阵。
- top()：在不移除的情况下，返回堆栈最顶部矩阵的引用。
- <stack>.top() *= rotate（构建旋转矩阵的参数）。
- <stack>.top() *= scale（构建缩放矩阵的参数）。
- <stack>.top() *= translate（构建平移矩阵的参数）。

（以上三项）直接对堆栈顶部的矩阵应用变换。

如前面列表中所示，“*=”运算符在 mat4 中被重载，因此它可以用于连接矩阵。因此，我们通常将它用于向矩阵堆栈顶部的矩阵添加平移、旋转等，正如我们展示出来的这些形式。

现在，我们不再通过创建 mat4 的实例来构建变换，而是使用 push()命令在堆栈顶部创建新的矩阵。然后再根据需要将期望的变换应用于堆栈顶部的新创建的矩阵。

推入堆栈的第一个矩阵通常是视图矩阵。它上面的矩阵是复杂程度越来越高的模型-视图矩阵；也就是说，它们应用了越来越多的模型变换。这些变换既可以直接应用，也可以先结合其他矩阵。

在我们的行星系统示例中，位于视图矩阵正上方的矩阵将是太阳的 MV 矩阵。在它之上的矩阵将是地球的 MV 矩阵，由太阳的 MV 矩阵的副本和应用于其之上的地球模型矩阵变换组成。也就是说，地球的 MV 矩阵是通过将行星的变换结合到太阳的变换中而建立的。同样，月球的 MV 矩阵位于行星的 MV 矩阵之上，并通过将月球的模型矩阵变换应用于紧邻其下方的行星的 MV 矩阵来构建。

在渲染月球之后，可以通过从堆栈中“弹出”第一个月球的矩阵（将堆栈的顶部恢复到行星的模型-视图矩阵），然后重复第二个月球的过程，来渲染第二个“月球”。

基本方法如下。

（1）我们声明我们的堆栈，给它起名为“mvStack”。

（2）当相对于父对象创建新对象时，调用“mvStack.push(mvStack.top())”。

（3）应用新对象所需的变换，也就是将所需的变换乘以它。

（4）完成对象或子对象的绘制后，调用“mvStack.pop()”从矩阵堆栈顶部移除其模型-视图矩阵。

在后面的章节中，我们将学习如何创建球体并使它们看起来像行星和卫星。就目前而言，为了简单起见，我们将使用我们的金字塔和几个立方体构建一个“行星系统”。

表 4.1 概述了使用矩阵堆栈的 display()函数通常是什么结构。

表 4.1 使用矩阵堆栈的 display()函数的结构

配置	● 实例化矩阵堆栈
摄像机	● 将新矩阵推入堆栈（这将实例化一个空的视图矩阵） ● 将变换应用于堆栈顶部的视图矩阵

续表

父对象	● 将新矩阵推入堆栈（这将是父 MV 矩阵——对第一个父对象来说，它直接复制一份视图矩阵） ● 应用变换，将父对象的模型矩阵和复制的视图矩阵结合 ● 发送最顶层的矩阵（即对顶点着色器中的 MV 矩阵统一变量使用"glm::value_ptr()"） ● 绘制父对象
子对象	● 将新矩阵推入堆栈。这将是子对象的 MV 矩阵，最初直接复制一份父对象的 MV 矩阵 ● 应用变换，将子对象的模型矩阵和复制的父 MV 矩阵结合 ● 发送最顶层的矩阵（即对顶点着色器中的 MV 矩阵统一变量使用"glm::value_ptr()"） ● 绘制子对象
清理	● 将子对象的 MV 矩阵弹出堆栈 ● 将父对象的 MV 矩阵弹出堆栈 ● 将视图矩阵弹出堆栈

请注意，金字塔（“太阳”）绕自己的轴旋转是在它自己的局部坐标空间中，不影响“子对象”（这里指行星和月亮）。因此，“太阳”的旋转（如图 4.12 所示）被推到堆栈上，但是在绘制“太阳”之后，它必须从堆栈中移除（弹出）。

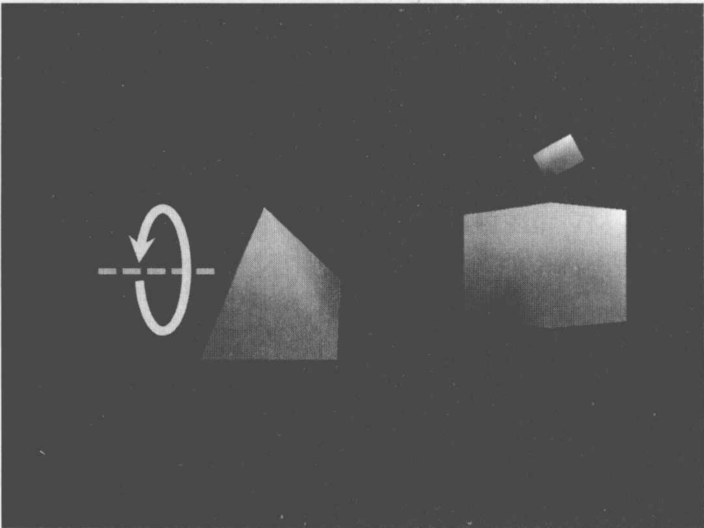


图 4.12 金字塔（“太阳”）的旋转

大立方体（行星）围绕太阳的旋转（如图 4.13（左）所示）将影响月球的运动，因此它被推到堆栈上并在绘制月球时保持在那里。相比之下，行星在其轴上的旋转（如图 4.13（右）所示）是局部的，不会影响月亮，因此在绘制月球之前它需要被从堆栈中弹出。

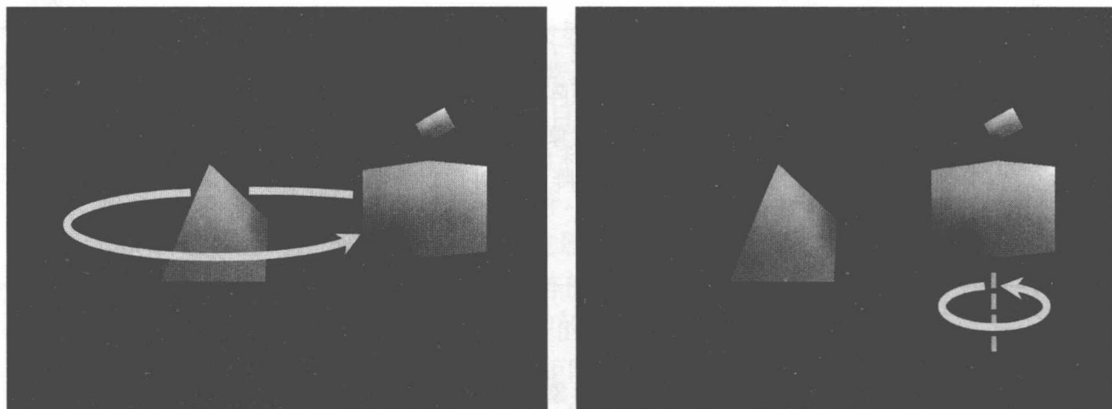


图 4.13 大立方体（行星）围绕太阳的旋转（左）和行星在其轴上的旋转（右）

类似地，我们会将变换推入堆栈以进行月球的旋转（围绕行星及其轴），如图 4.14 所示。

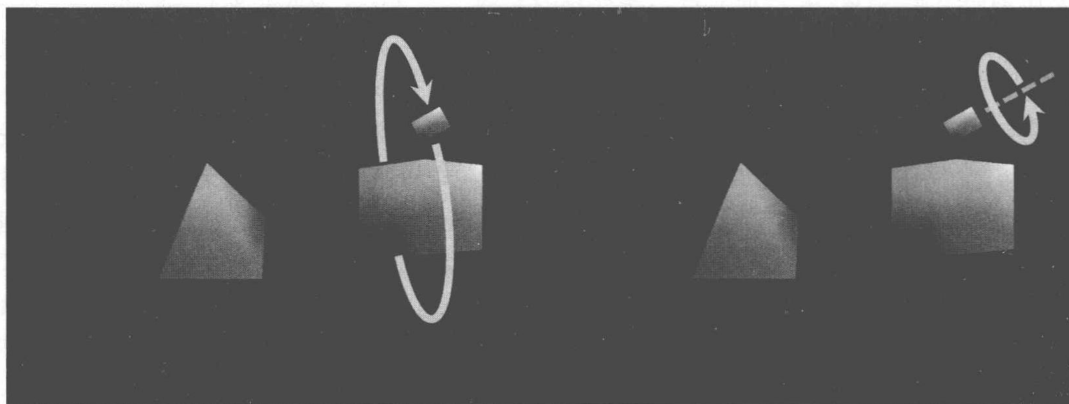


图 4.14 月球的旋转（围绕行星及其轴）

以下是“行星”的步骤顺序。

- push()将是行星 MV 矩阵中也会影响子对象的部分。
- translate(...)将太阳周围的行星运动结合到行星的 MV 矩阵中。在这个例子中，我们使用三角运算来计算行星运动的平移。
- push()将是行星的完整 MV 矩阵，也包括它的轴旋转。
- rotate(...)结合行星的轴旋转（稍后会弹出，不会影响子对象）。
- glm::value_ptr(mvStack.top())获取 MV 矩阵，然后将其发送到 MV 统一变量。
- 绘制星球。
- pop()将行星 MV 矩阵从堆栈中移除，暴露出它下面行星 MV 矩阵的不包括行星轴旋转的早期副本（因此只有行星的平移会影响月亮）。

现在我们可以编写完整的 display()函数，如程序 4.4 所示。

程序 4.4 使用矩阵堆栈的简单太阳系

```
stack<glm::mat4> mvStack;
void display(GLFWwindow* window, double currentTime) {
   // 配置背景、深度缓冲区、渲染程序，以及和原来一样的投影矩阵
```

```
. . .
// 将视图矩阵推入堆栈
vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
mvStack.push(vMat);

// ---------------------- 金字塔 == 太阳 ---------------------------------------------
mvStack.push(mvStack.top());
mvStack.top() *= glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f)); // 太阳位置
mvStack.push(mvStack.top());
mvStack.top() *= glm::rotate(glm::mat4(1.0f), (float)currentTime, glm::vec3(1.0f, 0.0f, 0.0f));
                                                                              // 太阳旋转
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
glEnable(GL_DEPTH_TEST);
glEnable(GL_LEQUAL);
glDrawArrays(GL_TRIANGLES, 0, 18);     // 绘制太阳
mvStack.pop();                         // 从堆栈中移除太阳的轴旋转

//----------------------- 立方体 == 行星 ---------------------------------------------
mvStack.push(mvStack.top());
mvStack.top() *=
   glm::translate(glm::mat4(1.0f), glm::vec3(sin((float)currentTime)*4.0, 0.0f, cos((float)
      currentTime)*4.0));
mvStack.push(mvStack.top());
mvStack.top() *= glm::rotate(glm::mat4(1.0f), (float)currentTime, glm::vec3(0.0, 1.0, 0.0));
                                                                          // 行星旋转
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 36);      // 绘制行星
mvStack.pop();                          // 从堆栈中移除行星的轴旋转

//----------------------- 小立方体 == 月球 ------------------------------------
mvStack.push(mvStack.top());
mvStack.top() *=
  glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, sin((float)currentTime)*2.0,
                                         cos((float)currentTime)*2.0));
mvStack.top() *= glm::rotate(glm::mat4(1.0f), (float)currentTime, glm::vec3(0.0, 0.0, 1.0));
                                                            // 月球旋转
mvStack.top() *= glm::scale(glm::mat4(1.0f), glm::vec3(0.25f, 0.25f, 0.25f)); // 让月球小一些
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 36);        // 绘制月球

// 从堆栈中移除月球缩放、旋转、位置矩阵，行星位置矩阵，太阳位置矩阵，和视图矩阵
mvStack.pop(); mvStack.pop(); mvStack.pop(); mvStack.pop();
}
```

矩阵堆栈操作已突出显示。有几个值得注意的细节：

- 我们在模型矩阵中引入了缩放操作。我们希望月球比行星更小，所以我们在为月球构建 MV 矩阵时调用了 scale()。
- 在这个例子中，我们使用三角运算 sin()和 cos()来计算行星绕太阳的旋转（作为平移的方式），以及月球绕行星的旋转。

- 两个缓冲区#0和#1分别包含立方体和金字塔的顶点。
- 注意在glUniformMatrix()命令中调用的glm::value_ptr(mvMatrix.top())函数。这个调用获取了堆栈顶部矩阵中的值，然后将这些值发送到统一变量（在本例中为太阳、行星以及月球的MV矩阵）。

此处省略顶点和片段着色器代码——它们与前一个示例相同。我们还移动了金字塔（“太阳”）和摄像机的初始位置，以使场景在屏幕上居中。

4.9 应对“Z冲突”伪影

回想一下，在渲染多个对象时，OpenGL使用Z缓冲区算法（Z-buffer algorithm）（如图2.14所示）来进行隐藏面消除。通常情况下，通过选择最接近相机的相应片段的颜色作为像素的颜色，这种方法解决了哪些物体的曲面可见并呈现到屏幕，而哪些曲面位于其他物体后面因此不应该被渲染。

然而，有时候场景中的两个物体表面重叠并位于重合的平面中，这使得Z缓冲区算法难以确定应该渲染两个表面中的哪一个（因为两者都不“最接近”摄像机）。发生这种情况时，浮点舍入误差可能会导致渲染表面的某些部分使用其中一个对象的颜色，而其他部分则使用另一个对象的颜色。这种不自然的伪影被称为Z冲突（Z-fighting）或深度冲突（Depth-fighting），因为这种效果是渲染的片段在Z缓冲区中相互对应的像素条目上“冲突斗争”的结果。图4.15（见彩插）显示了两个具有重叠重合（顶）面的盒子之间的Z冲突示例。

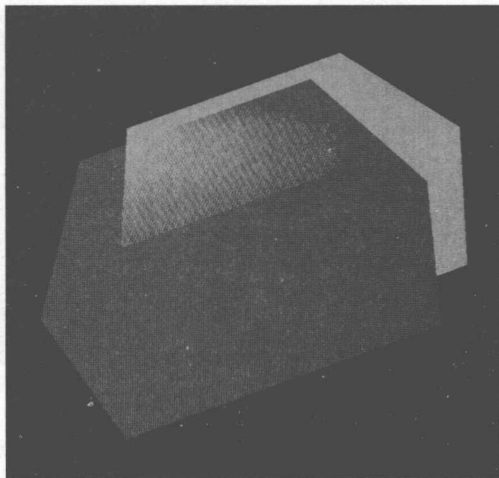


图4.15　Z冲突示例

创建地形或阴影时经常会出现这种情况。在这种情况下，有时Z冲突是可以预知的，并且校正它的常用方法是稍微移动一个物体，使得表面不再是共面的。我们将在第8章中看到这样的例子。

Z冲突还可能是由于深度缓冲器中的值的精度有限。对于由Z缓冲器算法处理的每个像素，其深度信息的精度受深度缓冲器中可存储的位数限制。用于构建透视矩阵的近剪裁平面和远剪裁平面之间的范围越大，具有相似（但不相等）的实际深度的两个对象的点在深度缓冲区中的数值表示越可能相同。因此，程序员可以选择适当的近、远剪裁平面值来最小化两个平面之间的距离，同时仍然确保场景必需的所有对象都位于视锥内。

同样重要的是要理解，由于透视变换的影响，改变近剪裁平面值可能比对远剪裁平面进行等效变化对于Z冲突伪影具有更大的影响。因此，建议避免选择太靠近眼睛的近剪裁平面。

本书前面的例子只是简单地使用了0.1和1000的值（在我们对perspective()的调用中）

用于近剪裁平面和远剪裁平面。这些可能需要针对您的场景进行调整。

4.10 图元的其他选项

OpenGL 支持许多图元类型——到目前为止我们已经看到了两个：GL_TRIANGLES 和 GL_POINTS。事实上，还有好几个其他的选择。OpenGL 支持的所有可用图元类型都属于三角形、线、点或者补丁的类别。以下是一个完整的清单。

三角形图元：

GL_TRIANGLES	本书中常见的图元类型。管线中传递的每 3 个顶点数据组成一个三角形： 0 1 2 / 3 4 5 / 6 7 8 顶点： ✓ ✓ ✓ 等 三角形：
GL_TRIANCLE_STRIP	管线中传递的每个顶点实际上和之前的两个顶点组成一个三角形： 顶点： 0 1 2 3 4 三角形： ✓ ✓ ✓ 等
GL_TRIANGLE_FAN	管线中传递的每对顶点和最开始的第一个顶点组成一个三角形： 顶点： 0 1 2 3 4 等 三角形：
GL_TRIANGLES_ADJACENCY	仅用于几何着色器。允许着色器访问当前三角形的顶点，以及额外的相邻顶点
GL_TRIANGLE_STRIP_ADJACENCY	仅用于几何着色器。类似 GL_TRIANGLES_ADJACENCY，除了三角形顶点像 GL_TRIANGLE_STRIP 中一样互相重叠

线图元：

GL_LINES	管线中传递的每两个顶点组成一条线： 顶点： 0 1 / 2 3 / 4 5 线： ✓ ✓ ✓ 等

续表

GL_LINE_STRIP 0 1 2 3	管线中传递的每个顶点和前一个顶点组成一条线： 顶点： 0 1 2 3 线： ✓ ✓ ✓ 等
GL_LINE_LOOP	跟 GL_LINE_STRIP 一样，除了第一个顶点和最后一个顶点之间也会组成一条线
GL_LINES_ADJACENCY	仅用于几何着色器。允许着色器访问当前线的顶点，以及额外的相邻顶点
GL_LINE_STRIP_ADJACENCY	类似 GL_LINES_ADJACENCY，除了线顶点像 GL_LINE_STRISTRIP 中一样互相重叠

点图元：

GL_POINTS	管线中传递的每个顶点是一个点

补丁图元：

GL_PATCH	仅用于细分着色器。指示一组顶点从顶点着色器传递到细分控制着色器，在这里它们通常用于将曲面细分网格成形为曲面

4.11 性能优先的编程方法

随着 3D 场景的复杂性增加，我们将越来越关注性能。我们已经看到一些例子，我们为了速度做出一些编程决策——例如当我们使用实例化时，以及当我们将昂贵的计算转移到着色器时。

实际上，我们展示的代码已经包含了一些我们尚未讨论的其他优化。我们现在来探索这些和其他重要技术。

4.11.1 尽量减少动态内存空间分配

考虑到性能，我们的 C++代码的最关键部分显然是 display()函数。这是在任何动画或实时渲染过程中重复调用的函数，因此在此函数中（或在它调用的任何函数中）我们必须努力实现最高的效率。

将 display()函数的开销保持在最低限度的一个重要方法是避免任何需要内存分配的步骤。因此，明显要避免的事情的例子包括：

- 实例化对象；
- 声明变量。

如果读者回顾我们迄今为止开发的程序，可以观察到，我们实际上在调用 display()函数之前就已经声明了 display()函数中使用到的每个变量，并分配了它的空间。声明或实例化几

乎从不出现在 display()函数中。例如，程序 4.1 在它开头包含以下代码块：

```
// 分配在 display()函数中使用的变量空间，这样它们就不必在渲染过程中分配
GLuint mvLoc, projLoc;
int width, height;
float aspect;
glm::mat4 pMat, vMat, mMat, mvMat;
```

请注意，我们故意在代码块的顶部放了一个注释，说明这些变量是预先分配的，以便稍后在 display()函数中使用（尽管我们到现在才明确地指出这一点）。

在我们的矩阵堆栈示例中发生了一个未预先分配的变量的情况。使用 C++堆栈类，每次“推”操作都会导致动态内存分配。有趣的是，在 Java 中，JOML 库提供了一个与 OpenGL 一起使用的 MatrixStack 类，它允许为矩阵堆栈预先分配空间！我们在本书的 Java 版中使用它。

还有其他更微妙的例子。例如，将数据从一种类型转换为另一种类型的函数调用在某些情况下可能会实例化并返回新转换的数据。因此，理解从 display()调用的任何库函数的行为非常重要。数学库 GLM 并没有专门针对速度优化设计。这导致一些操作可能引起动态内存分配。如果可能的话，我们会尽量使用直接在已经分配了空间的变量上操作的 GLM 函数。我们鼓励读者在性能至关重要时探索替代方法。

4.11.2 预先计算透视矩阵

可以减少 display()函数开销的另一个优化是将透视矩阵的计算移动到 init()函数中。我们在 4.5 节中提到了这种可能性（在脚注中）。虽然这很容易做到，但可能会有一点轻微的复杂情况。虽然通常并不需要重新计算透视矩阵，但是如果运行应用程序的用户调整窗口大小（例如通过拖动窗口角大小调整手柄），则重新计算就是必要的。

幸运的是，GLFW 可以配置在调整窗口大小时自动回调指定的函数。在调用 init()之前，我们将以下内容添加到 main()：

```
glfwSetWindowSizeCallback(window, window_reshape_callback);
```

第一个参数是 GLFW 窗口，第二个参数是 GLFW 在调整窗口大小时调用的函数的名称。然后，我们将计算透视矩阵的代码移动到 init()中，同时将其复制到名为 window_reshape_callback()的新函数中。

在程序 4.1 的例子中，如果我们重新组织代码，从 display()中删除透视矩阵的计算，那么 main()、init()、display()和新函数 window_reshape_callback()修改后的版本将如下所示。

```
void init(GLFWwindow* window) {
    . . .
    // 和之前版本一样，再加上以下三行代码：
    glfwGetFramebufferSize(window, &width, &height);
    aspect = (float)width / (float)height;
    pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);      // 1.0472 radians = 60 degrees
}

void display(GLFWwindow* window, double currentTime) {
    . . .
```

```
    // 和之前版本一样，再移除以下几行代码：

    // build perspective matrix
    glfwGetFramebufferSize(window, &width, &height);
    aspect = (float)width / (float)height;
    pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);
    // 函数余下部分没有变化
    . . .
}

void window_reshape_callback(GLFWwindow* window, int newWidth, int newHeight) {
    aspect = (float)newWidth / (float)newHeight;    // 回调提供的新的宽度、高度
    glViewport(0, 0, newWidth, newHeight);          // 设置和帧缓冲区相关的屏幕区域
    pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);
}

int main(void) {
    . . .
    // 和之前版本一样，再加上以下调用：

    glfwSetWindowSizeCallback(window, window_reshape_callback);
    init(window)
    while (!glfwWindowShouldClose(window)) {
        // 余下和以前一样
    }
```

本书配套资源中的程序，与透视矩阵计算有关的实现都是以这种方式组织的，从程序4.1的颜色插值版本开始。

4.11.3 背面剔除

提高渲染效率的另一种方法是利用OpenGL的*背面剔除*能力。当3D模型完全“闭合”时，意味着内部永远不可见（例如对于立方体和金字塔），那么外表面的那些与观察者背离呈一定角度的部分将始终被同一模型的其他部分遮挡。也就是说，那些背离观察者的三角形不可能被看到（无论如何它们都会在隐藏面消除的过程中被覆盖），因此没有理由光栅化或渲染它们。

我们可以使用命令glEnable(GL_CULL_FACE)要求OpenGL识别并“剔除”（不渲染）背向的三角形。我们还可以使用glDisable(GL_CULL_FACE)禁用背面剔除。默认情况下，背面剔除是关闭的，因此如果您希望OpenGL剔除背向三角形，必须手动启用它。

启用背面剔除时，默认情况下，只有三角形朝前时才会被渲染。此外，默认情况下，如果三角形的3个顶点从OpenGL摄像机中查看是以逆时针顺序排列的（基于它们在缓冲区中定义的顺序），则三角形被视为面向前方。顶点沿顺时针方向排列的三角形（从OpenGL摄像机中看）是朝后的，不会被渲染。这种逆时针方向定义的“前向”有时被称为*缠绕顺序*，可以使用函数调用glFrontFace(GL_CCW)显式设置逆时针（默认）为正向，或glFrontFace(GL_CW)设置顺时针为正向。类似地，也可以显式设置是否渲染正向或背向的三角形。实际上，为了这个目的，我们指定哪些不被渲染——即哪些被“剔除”。我们可以通过调用glCullFace(GL_BACK)指定面向背面的三角形被剔除（尽管这是不必要的，因为它是默认的）。或者，我们可以通过分别用GL_FRONT或GL_FRONT_AND_BACK替换参数GL_BACK来指定剔除前向三角形，甚至剔除所有三角形。

正如我们将在第 6 章中看到的那样，3D 模型通常被设计成外表面由相同缠绕顺序的三角形构成——最常见的是逆时针——因此如果启用剔除，则默认情况下模型的外部面向相机的表面部分会被渲染。因为默认情况下 OpenGL 假定的缠绕顺序是逆时针方向，如果模型设计缠绕顺序为顺时针方向，那么如果启用了背面剔除，需要由程序员调用 gl_FrontFace (GL_CW)来解决此问题。

注意，在 GL_TRIANGLE_STRIP 的情况下，每个三角形的缠绕顺序不停地互换。OpenGL 通过在构建每个连续三角形时“翻转”顶点序列来补偿这一点，如下所示：0-1-2，然后 2-1-3、2-3-4、4-3-5、4-5-6 等。

背面剔除通过确保 OpenGL 不花时间光栅化和渲染从不被看到的表面来提高性能。我们在本章中看到的大多数示例都非常小，以至于没有动机进行背面剔除（图 4.9 中展示了一个例外，其中包含了 100 000 个多边形动画实例，这可能会对某些系统造成性能挑战）。在实践中，大多数 3D 模型通常是“闭合的”，因此习惯上会常规地启用背面剔除。例如，我们可以通过修改 display()函数向程序 4.3 添加背面剔除，如下所示。

```
void display(GLFWwindow* window, double currentTime) {
    . . .
    glEnable(GL_CULL_FACE);

    // 绘制立方体
    . . .
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glFrontFace(GL_CW);                    // 立方体顶点的缠绕顺序为顺时针方向
    glDrawArrays(GL_TRIANGLES, 0, 36);

    // 绘制金字塔
    . . .
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glFrontFace(GL_CCW);                   // 金字塔顶点缠绕顺序为逆时针方向
    glDrawArrays(GL_TRIANGLES, 0, 18);
}
```

使用背面剔除时，正确设置缠绕顺序非常重要。不正确的设置，例如当应该设置 GL_CCW 时设置成了 GL_CW，可能会导致渲染出对象的内部而不是其外部，这就会产生类似于不正确的透视矩阵的失真。

效率不是进行背面剔除的唯一原因。在后面的章节中，我们将看到其他用途，例如我们想要查看 3D 模型内部或使用透明度时的情况。

补充说明

在 OpenGL/GLSL 中，有许多其他功能和结构可用于管理和利用数据，我们在本章中仅涉及了很浅层的一部分。例如，我们没有描述统一块，这是一种类似于 C 中的 struct 的用于统一变量的机制。甚至可以设置统一块从缓冲区接收数据。另一个强大的机制是**着色器存储块**，它本质上是一个着色器可以写入的缓冲区。

关于管理数据的许多选项的一个很好的参考资料是《OpenGL 超级宝典》[SW15]，特别是关于数据的章节（第 7 版的第 5 章）。它还描述了我们所涵盖的各种命令的许多细节和

选项。本章的前两个示例程序，即程序 4.1 和程序 4.2 受到《OpenGL 超级宝典》中类似示例的启发。

我们还需要学习如何管理其他类型的数据，以了解如何将它们发送给 OpenGL 管线。其中之一是**纹理**，包含可用于“绘制”场景中对象的彩色图像数据（像照片）。我们将在第 5 章中研究纹理图像。我们将进一步研究的另一个重要缓冲区是**深度缓冲区**（或者叫 Z 缓冲区）。当我们在第 8 章中研究阴影时，这将变得很重要。关于如何在 OpenGL 中管理图形数据，我们还有很多知识需要学习！

习题

4.1 （项目）修改程序 4.1 以使用你自己设计的其他简单 3D 形状替换立方体。请务必在 glDrawArrays()命令中正确指定顶点数。

4.2 （项目）在程序 4.1 中，在 display()函数中“view”矩阵被简单地定义为摄像机位置的负数：

```
vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
```

将此代码替换为图 3.13 所示的计算实现。这将允许你通过指定摄像机位置和 3 个朝向轴来定位摄像机。你将发现有必要存储 3.7 节中描述的向量 ***U***、***V***、***N***。然后，尝试不同的摄像机视点，并观察渲染立方体的最终外观。

4.3 （项目）修改程序 4.4 以包含第二个“行星”，使用练习 4.1 中你自定义的 3D 形状。确保你的新“行星”处于与现有行星不同的轨道上，这样它们就不会发生碰撞。

4.4 （项目）修改程序 4.4，使用“查看”函数构建“视图”矩阵（如第 3.9 节所述）。然后尝试将“查看”参数设置到不同的位置，例如查看太阳（在这种情况下场景应该看起来正常），查看行星或查看月球。

4.5 （研究）提出 glCullFace(GL_FRONT_AND_BACK)的实际用途。

参考资料

[BL16] Blender, The Blender Foundation, accessed October 2018.

[HT16] J. Hastings-Trew, JHT's Planetary Pixel Emporium, accessed October 2018.

[MA16] Maya, AutoDesk, Inc., accessed October 2018.

[NA16] NASA 3D Resources, accessed October 2018.

[OL16] Legacy OpenGL, accessed July 2016.

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

第5章 纹理贴图

纹理贴图是在光栅化的模型表面上覆盖图像的技术。它是为渲染场景添加真实感的最基本和最重要的方法之一。

纹理贴图非常重要，因此硬件也为它提供了支持，使得它具备了实现实时的照片级真实感的超高性能。纹理单元是专为纹理设计的硬件组件，现代显卡通常带有数个纹理单元。

5.1 加载纹理图像文件

为了在 OpenGL/GLSL 中有效地完成纹理贴图，需要协调好以下几个不同的数据集和机制：

- 用于保存纹理图像的纹理对象（在本章中我们仅考虑 2D 图像）；
- 一个特殊的统一采样器变量，以便顶点着色器可以访问纹理；
- 用于保存纹理坐标的缓冲区；
- 用于将纹理坐标传递给管线的顶点属性；
- 显卡上的纹理单元。

纹理图像可以是任何图像。它可以是人造的或者自然产生的事物的图片，例如布、草或行星表面；它也可以是几何图样，例如图 5.1 中的棋盘图样。在电子游戏和动画电影中，纹理图像通常用于给角色绘制面部和衣服，或者在像图 5.1 中的海豚等生物身上绘制皮肤。

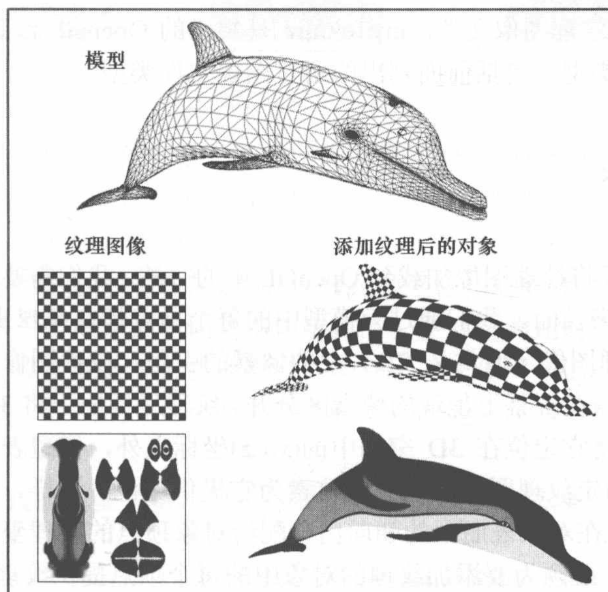


图 5.1　使用两张不同的图像给同一个海豚模型添加纹理[TU16]

图像通常存储在图像文件中，例如.jpg、.png、.gif 或.tiff 格式。为了使纹理图像可以被用于 OpenGL 管线中的着色器，我们需要从图像中提取颜色并将它们放入 OpenGL 纹理对象（用于保存纹理图像的内置 OpenGL 结构）中。

许多 C++库可用于读取和处理图像文件，常见的选择包括 Cimg、BoostGIL 和 Magick++。我们选择使用专为 OpenGL 设计的名为 SOIL2 [SO17]的库，它基于曾经非常流行但现在已经过时的库 SOIL。在附录 A 和附录 B 中介绍了 SOIL2 的安装步骤。

通常我们将纹理加载到 OpenGL 应用程序的步骤是：（a）使用 SOIL2 实例化 OpenGL 纹理对象并从图像文件中读入数据；（b）调用 glBindTexture()以使新创建的纹理对象处于激活状态；（c）使用 glTexParameter()函数调整纹理设置。最终获得的结果就是现在可用的 OpenGL 纹理对象的整型 ID。

创建一个纹理对象，首先需要声明一个 GLuint 类型的变量。正如我们所看到的，这是一个用于保存 OpenGL 对象的整型 ID 引用的 OpenGL 类型。接下来，我们调用 SOIL_load_OGL_texture()来实际生成纹理对象。SOIL_load_OGL_texture()函数接受图像文件名作为其参数之一（稍后将描述一些其他参数）。这些步骤在以下函数中实现：

```
GLuint loadTexture(const char *texImagePath) {
  GLuint textureID;
  textureID = SOIL_load_OGL_texture(texImagePath,
         SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_INVERT_Y);
  if (textureID == 0) cout << "could not find texture file" << texImagePath << endl;
  return textureID;
}
```

我们会经常使用这个函数，所以我们将它添加到 Utils.cpp 实用工具类中。这样，我们的 C++应用程序就只需调用上述的 loadTexture()函数来创建 OpenGL 纹理对象，如下所示。

```
GLuint myTexture = Utils::loadTexture("image.jpg");
```

其中 image.jpg 是纹理图像文件，myTexture 是生成的 OpenGL 纹理对象的整型 ID。这里支持多种图像文件类型，包括前面列出的所有图像文件类型。

5.2 纹理坐标

现在我们已经有了将纹理图像加载到 OpenGL 中的方法，我们需要指定我们希望如何将纹理应用于对象的渲染表面。我们通过为模型中的每个顶点指定纹理坐标来完成此操作。

纹理坐标是对纹理图像（通常是 2D）中的像素的引用。纹理图像中的像素被称为纹素（Texel），以便将它们与在屏幕上呈现的像素区分开。纹理坐标用于将 3D 模型上的点映射到纹理中的位置。除了将它定位在 3D 空间中的(x,y,z)坐标之外，模型表面上的每个点还具有纹理坐标(s,t)，用来指定纹理图像中的哪个纹素为它提供颜色。这样，物体的表面被按照纹理图像"涂画"。纹理在对象表面上的朝向由分配给对象顶点的纹理坐标来确定。

要使用纹理贴图，必须为要添加纹理的对象中的每个顶点提供纹理坐标。OpenGL 将使用这些纹理坐标，查找存储在纹理图像中的引用的纹素的颜色，来确定模型中每个光栅化

像素的颜色。为了确保渲染模型中的每个像素都使用纹理图像中的适当纹素进行绘制，纹理坐标也需要被放入顶点属性中，以便它们也由光栅着色器进行插值。以这种方式，纹理图像与模型顶点一起被插值或者填充。

对于通过顶点着色器的每组顶点坐标(*x*,*y*,*z*)，会有一组相应的纹理坐标(*s*,*t*)。因此，我们将设置两个缓冲区，一个用于顶点（每个条目中有 3 个分量 *x*、*y* 和 *z*），另一个用于相应的纹理坐标（每个条目中有两个分量 *s* 和 *t*）。这样，每次顶点着色器的调用接收到一个顶点的数据，现在包括了其空间坐标和相应的纹理坐标。

2D 纹理坐标最为常见（OpenGL 确实支持其他一些维度，但我们不会在本章中介绍它们）。2D 纹理图像被设定为矩形，左下角的位置坐标为(0,0)，右上角的位置坐标为(1,1)。[①]理想情况下，纹理坐标应该在[0…1]范围内取值。

考虑图 5.2 中的示例。回想一下，立方体模型由三角形构成。我们的示意图中突出显示了立方体一侧的 4 个角，但请记住，立方体的每个正方形侧面需要两个三角形。指定这一个立方体侧面的 6 个顶点中的每一个的纹理坐标沿着 4 个角列出，左上角和右下角各自由一对顶点组成。示例里也显示了纹理图像。纹理坐标（由 *s* 和 *t* 描述）将图像的部分（纹素）映射到模型正面的光栅化像素上。请注意，顶点之间的所有中间像素都已使用图像中间插值的纹素进行绘制。这正是因为纹理坐标在顶点属性中被发送到片段着色器，因此也像顶点本身一样被插值。

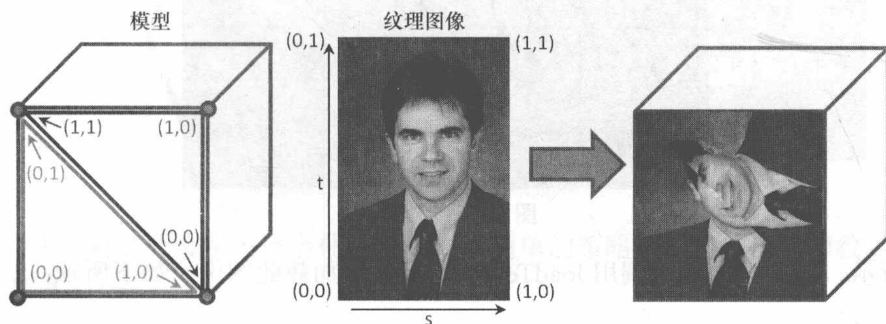


图 5.2 纹理坐标

在这个示例中，出于说明的目的，我们故意指定了会导致奇怪的表面绘制的纹理坐标。仔细观察，您还可以看到图像看起来略微拉伸——这是因为纹理图像的长宽比与立方体面相关的给定纹理坐标的长宽比不匹配。

对于立方体或金字塔这样的简单模型，选择纹理坐标相对容易。但对于具有大量三角形的更复杂的弯曲模型，手动确定它们是不切实际的。在弯曲的几何形状（例如球形或环面）的情况下，可以通过算法或数学方式计算纹理坐标。对于使用 Maya [MA16]或 Blender [BL16]等建模工具构建的模型，这些工具提供有“UV 映射”功能（在本书范围之外），使得这项任务更容易。

让我们回去渲染我们的金字塔，只是这次用砖的图像添加纹理。我们需要指定：（a）引

① 这是 OpenGL 纹理对象所采用的方向。然而，这与存储在许多标准图像文件格式中的图像的方向不同，在那些图像中原点位于左上角。我们通过指定 SOIL_FLAG_INVERT_Y 参数，垂直翻转图像来重新定向，使其与 OpenGL 的预期格式相对应，就像我们在 loadTexture()函数中对 SOIL_load_OGL_texture()进行的调用一样。

用纹理图像的整型 ID；(b) 模型顶点的纹理坐标；(c) 用于保存纹理坐标的缓冲区；(d) 顶点属性，以便顶点着色器可以接收并通过管线转发纹理坐标；(e) 显卡上用于保存纹理对象的纹理单元；(f) 我们将很快看到的用于访问 GLSL 中纹理单元的统一采样器变量。这些将在下一节中描述。

5.3 创建纹理对象

假设此处显示的纹理图像（如图 5.3 所示）存储在名为“brick1.jpg”[LU16]的文件中。

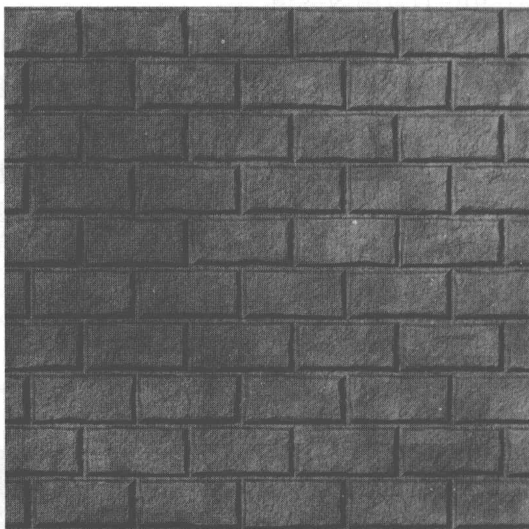


图 5.3 纹理图像

如前所示，我们可以通过调用 loadTexture()函数来加载此图像，如下所示：

```
GLuint brickTexture = Utils::loadTexture("brick1.jpg");
```

回想一下，纹理对象由整型 ID 标识，因此 brickTexture 的类型为 GLuint。

5.4 构建纹理坐标

我们的金字塔有 4 个三角形侧面和底部的正方形底面。虽然在几何上这只需要 5 个点，但我们得用三角形来渲染它。这需要 4 个三角形用于侧面，以及 2 个三角形用于正方形底面，总共 6 个三角形。每个三角形有 3 个顶点，必须在模型中指定总共 6×3 = 18 个顶点。

我们已经在程序 4.3 的浮点数组 pyramidPositions[ ]中列出了金字塔的几何顶点。我们可以通过多种方式定位纹理坐标，以便将砖纹理绘制到金字塔上。一种简单（尽管不完美）的方法是使图像的顶部中心对应于金字塔的尖顶，如图 5.4 所示。

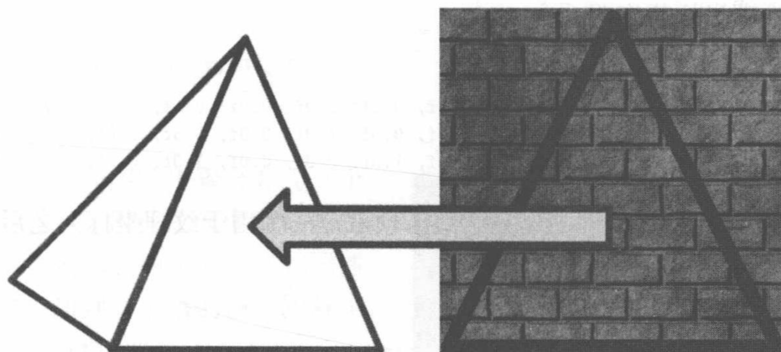


图 5.4 纹理图像的顶部中心对应金字塔的尖顶

我们可以为所有 4 个三角形侧面这样做。我们还需要绘制金字塔的正方形底面，它由 2 个三角形组成。一个简单而合理的方法是用图片中的整个区域为其添加纹理（图 5.5 所示的金字塔已被向后放倒，一个侧面朝下）。

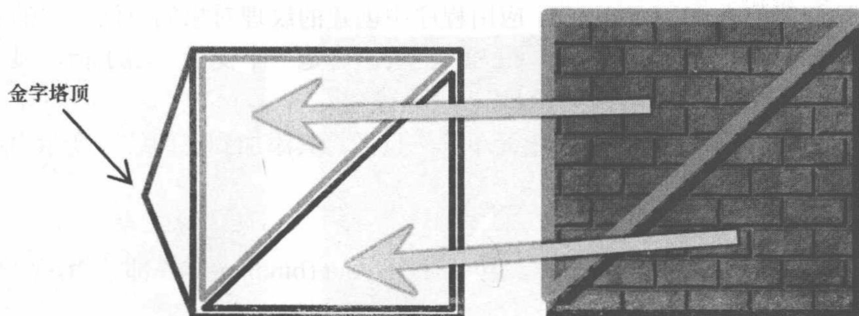


图 5.5 为金字塔底面添加纹理

对程序 4.3 中前 9 个金字塔顶点使用这个非常简单的策略，相应的顶点和纹理坐标数据组如图 5.6 所示。

```
        顶点                    纹理坐标
(-1.0,-1.0, 1.0)          (0,   0)       // 前侧面
( 1.0,-1.0, 1.0)          (1,   0)
(   0, 1.0,   0)          (.5,  1)
( 1.0,-1.0, 1.0)          (0,   0)       // 右侧面
( 1.0,-1.0,-1.0)          (1,   0)
(   0, 1.0,   0)          (.5,  1)
( 1.0,-1.0,-1.0)          (0,   0)       // 底面
(-1.0,-1.0,-1.0)          (1,   0)
(   0, 1.0,   0)          (.5,  1)
etc.
```

图 5.6 金字塔的纹理坐标（部分清单）

5.5 将纹理坐标载入缓冲区

我们可以用与前面加载顶点相似的方式将纹理坐标加载到 VBO 中。在 setupVertices()中，

我们添加以下纹理坐标值声明：

```
float pyrTexCoords[36] =
{ 0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,   0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,   // 前侧面、右侧面
  0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,   0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,   // 后侧面、左侧面
  0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f,   1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f }; // 底面的两个三角形
```

然后，在创建至少两个 VBO（一个用于顶点，一个用于纹理坐标）之后，我们添加以下代码行以将纹理坐标加载到 VBO #1 中：

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(pyrTexCoords), pyrTexCoords, GL_STATIC_DRAW);
```

5.6 在着色器中使用纹理：采样器变量和纹理单元

为了最大限度地提高性能，我们希望在硬件中执行纹理处理。这意味着我们的片段着色器需要一种访问我们在 C++/OpenGL 应用程序中创建的纹理对象的方法。它的实现机制是通过一个叫作统一采样器变量的特殊 GLSL 工具。这是一个变量，用于指示显卡上的纹理单元，从加载的纹理对象中提取或"采样"哪个纹素。

在着色器中声明一个采样器变量很简单——只需将其添加到您的统一变量中：

```
layout (binding=0) uniform sampler2D samp;
```

我们声明的变量名字叫作"samp"。声明的"layout (binding=0)"部分指定此采样器与纹理单元 0 相关联。

纹理单元（和相关的采样器）可用于对您希望的任何纹理对象进行采样，并且可以在运行时进行更改。您的 display()函数需要指定纹理单元要为当前帧采样的纹理对象。因此，每次绘制对象时，都需要激活纹理单元并将其绑定到特定的纹理对象，例如：

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, brickTexture);
```

可用纹理单元的数量取决于图形卡上提供的数量。根据 OpenGL API 文档，OpenGL 4.5 版要求每个着色器阶段至少有 16 个，所有阶段总共至少 80 个单元[OP16]。在这个例子中，我们通过在 glActiveTexture()调用中指定 GL_TEXTURE0，使得第 0 个纹理单元处于激活状态。

要实际执行纹理处理，我们需要修改片段着色器输出颜色的方式。以前，我们的片段着色器要么输出一个固定的颜色常量，要么从顶点属性获取颜色。相反，这次我们需要使用从顶点着色器（通过光栅着色器）接收的插值纹理坐标来对纹理对象进行采样，像这样调用 texture()函数：

```
in vec2 tc;             // 纹理坐标
. . .
color = texture(samp, tc);
```

5.7 纹理贴图：示例程序

程序 5.1 将前面介绍的步骤合并为一个程序。输出结果显示了用砖图像纹理贴图的金字塔，如图 5.7 所示。两个旋转（代码清单中未显示）被添加到金字塔的模型矩阵中以暴露金字塔的底面。

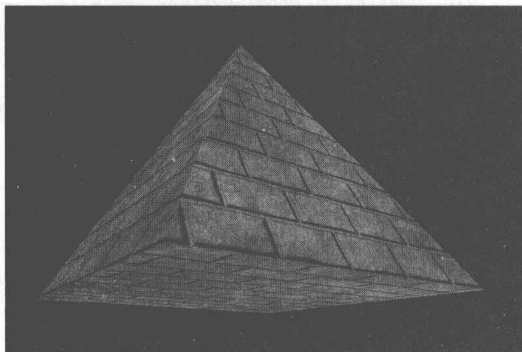


图 5.7 使用砖图像纹理贴图后的金字塔

现在，根据需要，通过更改 loadTexture() 调用中的文件名，将砖纹理图像替换为其他纹理图像是一件简单的事情。例如，如果我们用图像文件"ice.jpg"[LU16]替换"brick1.jpg"，我们得到的结果如图 5.8 所示。

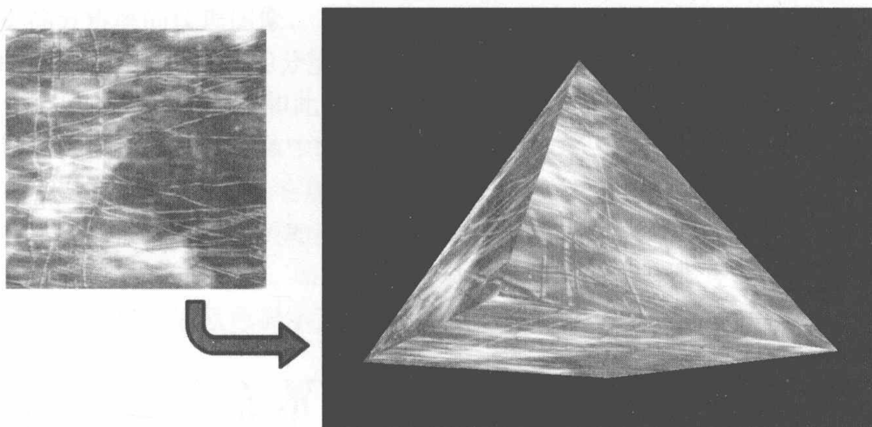


图 5.8 使用"冰"图像纹理贴图后的金字塔

程序 5.1 砖纹理的金字塔

```
C++/OpenGL 应用程序

#include <SOIL2/soil2.h>
// 其他#include 和以前一样
. . .
#define numVAOs 1
#define numVBOs 2

// 摄像机和对象位置、渲染程序、VAO 和 VBO 的变量和以前一样
. . .
// 显示函数的变量分配和以前一样
. . .
GLuint brickTexture;

void setupVertices(void) {
   float pyramidPositions[54] = { /* 如程序 4.2 中列出的数据 */

   float pyrTexCoords[36] = {
```

```
        0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,     0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,
        0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,     0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f,     1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f
    };

    // . . . 像以前一样生成VAO和至少两个VBO，并加载两个缓冲区：
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(pyramidPositions), pyramidPositions, GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(pyrTexCoords), pyrTexCoords, GL_STATIC_DRAW);
}

void init(GLFWwindow* window) {
    // 渲染程序配置、摄像机和对象位置没有改变
    . . .
    brickTexture = Utils::loadTexture("brick1.jpg");
}

void display(GLFWwindow* window, double currentTime) {
    . . .
    // 背景颜色配置、深度缓冲区、渲染程序，以及M、V、MV、PROJ矩阵没有变化
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, brickTexture);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    glDrawArrays(GL_TRIANGLES, 0, 18);
}

// main() 和以前一样
```

顶点着色器

```
#version 430
layout (location=0) in vec3 pos;
layout (location=1) in vec2 texCoord;
out vec2 tc;          // 纹理坐标输出到光栅着色器用于插值
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler2D samp;      // 顶点，着色器中未使用

void main(void)
{ gl_Position = proj_matrix * mv_matrix * vec4(pos,1.0);
  tc = texCoord;
}
```

片段着色器

```
#version 430
```

```
in vec2 tc;          // 输入插值过的材质坐标
out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler2D samp;

void main(void)
{ color = texture(samp, tc);
}
```

5.8 多级渐远纹理贴图

纹理贴图经常会在渲染图像中产生各种不期望的伪影。这是因为纹理图像的分辨率或长宽比很少与被纹理贴图的场景中区域的分辨率或长宽比相匹配。

当图像分辨率小于所绘制区域的分辨率时，会出现一种很常见的伪影。在这种情况下，需要拉伸图像以覆盖整个区域，就会变得模糊（并且可能变形）。根据纹理的性质，有时可以通过改变纹理坐标分配方式来对抗这种情况，使得纹理需要较少的拉伸。另一种解决方案是使用更高分辨率的纹理图像。

相反的情况是当图像纹理的分辨率大于被绘制区域的分辨率时。可能并不是很容易理解为什么这会造成问题，但确实如此！在这种情况下，可能会出现明显的叠影伪影，从而产生奇怪的错误图案，或移动物体中的“闪烁”效果。

叠影是由采样错误引起的。它通常与信号处理有关，不充分采样的信号被重建时，看起来会具有和实际不同的特性（例如波长）。例子如图 5.9 所示（见彩插）。原始波形显示为红色，沿波形的黄点代表采样点。如果采样点被用于重建波形，并且采样频率不足，则可能会定义出不同的波形（以蓝色显示）。

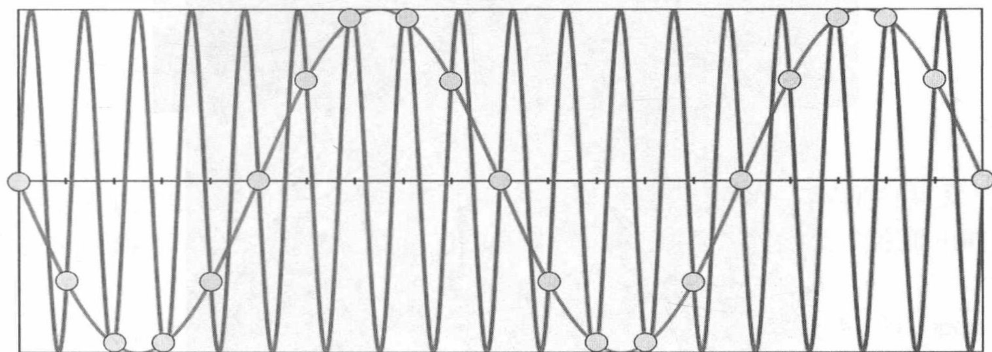


图 5.9 不充分采样造成的叠影

类似地，在纹理贴图中，当稀疏地采样高分辨率（和高细节）图像时（例如使用统一采样器变量时），提取到的颜色将不足以反映图像中的实际细节，而是可能看起来很随机。如果纹理图像具有重复图案，则叠影可能导致生成与原始图像不同的图案。如果被纹理贴图的对象正在移动，则纹素查找中的舍入误差可能导致给定纹理坐标处的采样像素的不断变化，从而在被绘制对象的表面上产生不希望的闪烁效果。

图 5.10 显示了一个立方体顶部的倾斜渲染特写，该立方体使用大尺寸高分辨率棋盘图像进行纹理贴图。

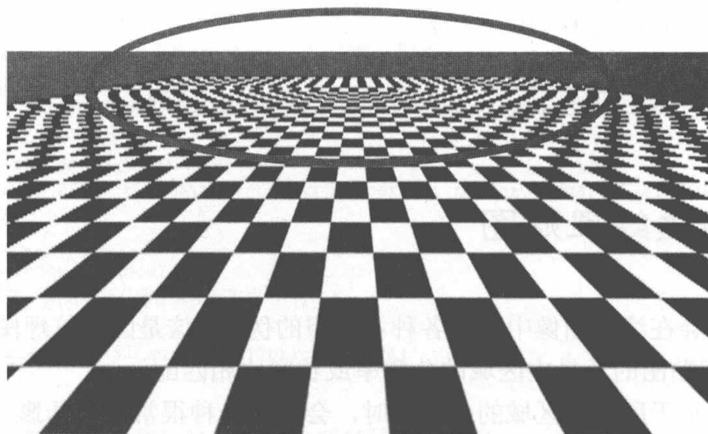


图 5.10 纹理贴图中的叠影

在图像顶部附近明显发生了混叠，棋盘的欠采样产生了“条纹”效果。虽然我们无法在静止图像中展示，但如果这是一个动画场景，则看起来的图案可能会在各种不正确的图案（包括图示的这一个在内）之间波动。

另一个例子如图 5.11 所示，其中的立方体已经使用月球表面的图像[HT16]进行纹理贴图。乍一看，这张图片显得清晰而细节丰富。然而，图像右上部分的某些细节是错误的，并且当立方体对象（或相机）移动时会导致“闪烁”。（不幸的是，我们无法在静止图像中清楚地显示闪烁效果。）

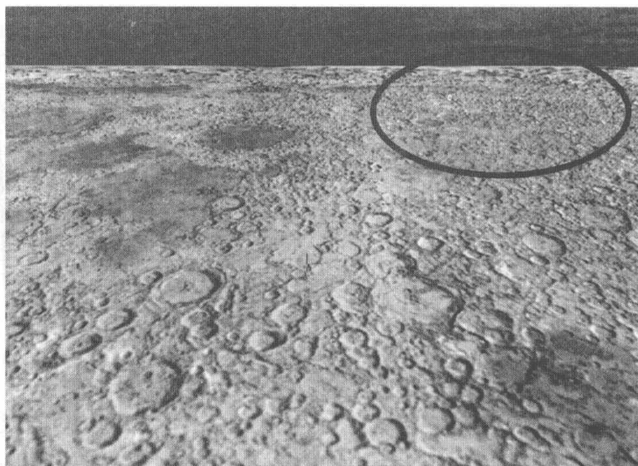


图 5.11 纹理贴图中的“闪烁”

使用多级渐远纹理贴图（Mipmapping）技术可以在很大程度上校正这一类的采样误差伪影，它需要用各种分辨率创建纹理图像的不同版本。然后，OpenGL 使用最适合正在处理的这一点处的分辨率的纹理图像进行纹理贴图。更好的是，可以为被贴图的区域使用最适合的分辨率的纹理图像的平均颜色。多级渐远纹理贴图应用于图 5.10 和图 5.11 中的图像的结果如图 5.12 所示。

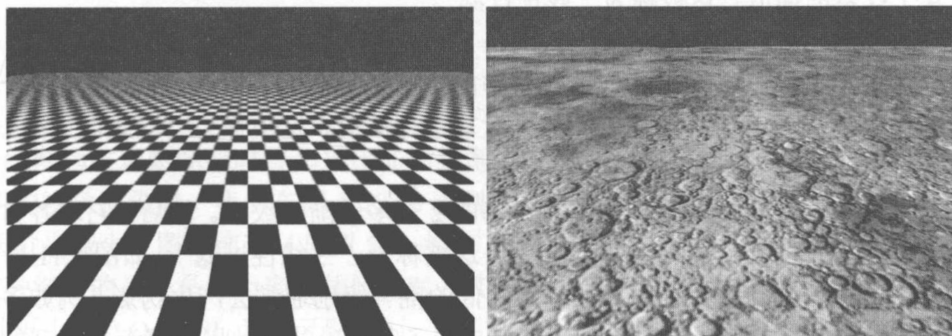


图 5.12　多级渐远纹理贴图结果

多级渐远纹理贴图通过一种巧妙的机制来工作，它在纹理图像中存储相同图像的连续的一系列较低分辨率的副本，所用的纹理图像比原始图像大 1/3。这是通过将图像的 *R*、*G* 和 *B* 分量分别存储在纹理图像空间的 3 个 1/4 中来实现的，然后在剩余的 1/4 图像空间中对于同一图像重复相当于原始分辨率 1/4 的处理。重复该细分直到剩余象限太小而不包含任何有用的图像数据。示例图像和生成的多级渐远纹理的可视化如图 5.13 所示（见彩插）。

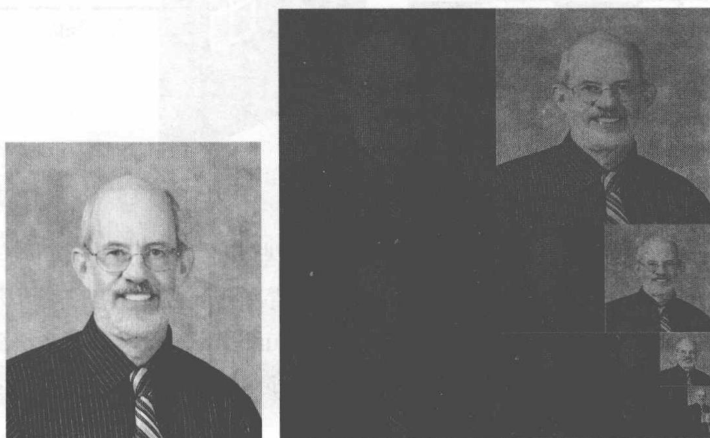


图 5.13　为图片生成多级渐远纹理

这种将几个图像填充到一个小空间中的方法（只比存储原始图像所需的空间大一点）是 Mipmapping 得名的原因。MIP 代表拉丁语 Multum In Parvo [WI83]，意思是“在很小的空间里有很多东西”。

实际给对象添加纹理时，可以通过多种方式对多级渐远纹理进行采样。在 OpenGL 中，可以通过将 GL_TEXTURE_MIN_FILTER 参数设置为所需的缩小方法来选择多级渐远纹理的采样方式，可以选取以下方法之一。

- GL_NEAREST_MIPMAP_NEAREST
  选择具有与纹素区域最相似的分辨率的多级渐远纹理。然后，它获得所需纹理坐标的最近纹素。
- GL_LINEAR_MIPMAP_NEAREST
  选择具有与纹素区域最相似的分辨率的多级渐远纹理。然后它取最接近纹理坐标的

4个纹素的插值。这被称为“线性过滤”。

- GL_NEAREST_MIPMAP_LINEAR
  选择具有与纹素区域最相似的分辨率的2个多级渐远纹理。然后，它从每个多级渐远纹理获取纹理坐标的最近纹素并对其进行插值。这被称为“双线性过滤”。
- GL_LINEAR_MIPMAP_LINEAR
  选择具有与纹素区域最相似的分辨率的2个多级渐远纹理。然后，它取各自最接近纹理坐标的4个纹素，并计算插值。这被称为“三线性过滤”，如图5.11所示。

三线性过滤通常是比较好的选择，因为较低的混合级别通常会产生伪影，例如多级渐远纹理级别之间的可见分离。图5.14显示了只启用了线性过滤的使用多级渐远纹理的棋盘的特写。请注意在多级渐远纹理的边界处垂直线突然从粗变为细（图中圈出的位置的伪影）。相比之下，图5.15中的示例使用了三线性过滤。

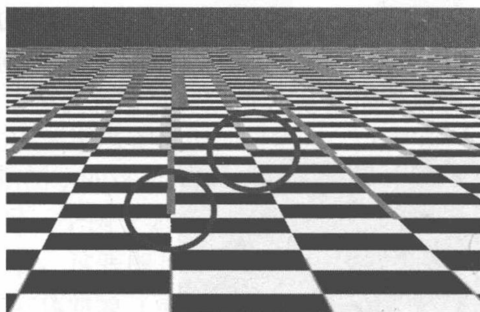


图5.14 线性过滤伪影

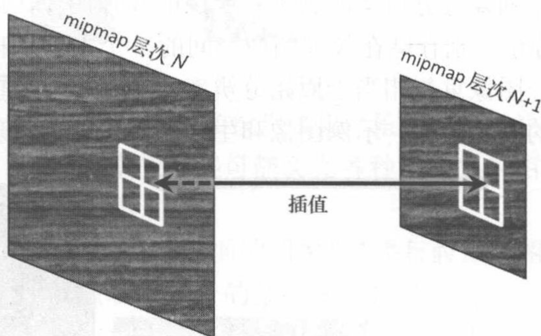


图5.15 三线性过滤

OpenGL提供了丰富的多级渐远纹理支持。有一些机制可用于构建你自己的多级渐远纹理级别，或者让OpenGL为你构建它们。在大多数情况下，OpenGL自动构建的多级渐远纹理已足够。这是通过将以下代码行添加进getTextureObject()函数之后立即执行的Utils::loadTexture()函数（前面的5.1节中介绍过）中实现的：

```
glBindTexture(GL_TEXTURE_2D, textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glGenerateMipmap(GL_TEXTURE_2D);
```

这通知OpenGL生成多级渐远纹理。使用glBindTexture()调用激活砖纹理，然后glTexParameteri()函数调用启用前面列出的缩小方法之一，例如上面调用中显示的GL_LINEAR_MIPMAP_LINEAR，它启用三线性过滤。

构建多级渐远纹理后，可以通过再次调用glTexParameteri()来更改过滤选项（尽管这很少需要），例如在display函数中。甚至可以通过选择GL_NEAREST或GL_LINEAR来禁用多级渐远纹理。

对于关键应用程序，可以使用您喜欢的任何图像编辑软件自行构建多级渐远纹理。然后可以通过为每个多级渐远纹理级别重复调用OpenGL的glTexImage2D()函数来创建纹理对象，并将它们添加为多级渐远纹理级别。对这种方法的进一步讨论超出了本书的范围。

5.9 各向异性过滤

多级渐远纹理贴图有时看起来比非多级渐远纹理贴图更模糊，尤其是当被贴图对象以严重倾斜的视角渲染时。我们在图 5.12 中看到了一个这样的例子，使用多级渐远纹理减少伪影的同时也减少了图像细节（与图 5.11 相比）。

这种细节的丢失是因为当物体倾斜时，其基元看起来沿一个轴（即宽度或高度）比沿另一个轴更小。当 OpenGL 为图元贴图时，它选择适合两个轴中较小的轴的多级渐远纹理（以避免“闪烁”伪影）。在图 5.12 中，表面远离观察者倾斜，因此每个渲染图元将使用适合其更小的高度的多级渐远纹理，这可能对其宽度来说分辨率太小了。

一种恢复一些丢失细节的方法是使用**各向异性过滤**（AF）。标准的多级渐远纹理贴图以各种正方形分辨率（例如 256 像素×256 像素、128 像素×128 像素等）对纹理图像进行采样，而 AF 却以多种矩形分辨率对纹理进行采样，例如 256 像素×128 像素、64 像素×128 像素等。这使得从各种角度观看并同时在纹理中保留尽可能多的细节成为可能。

各向异性过滤比标准多级渐远纹理贴图在计算上代价更高，并且不是 OpenGL 的必需部分。但是，大多数显卡都支持 AF（这被称为 OpenGL *扩展*），而 OpenGL 确实提供了一种查询显卡是否支持 AF 的方法，以及一种访问 AF 的方法。生成多级渐远纹理贴图后立即添加代码：

```
. . .
// 如果启用多级渐远纹理贴图
glBindTexture(GL_TEXTURE_2D, textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glGenerateMipmap(GL_TEXTURE_2D);

// 如果还启用各向异性过滤
if (glewIsSupported("GL_EXT_texture_filter_anisotropic")) {
    GLfloat anisoSetting = 0.0f;
    glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &anisoSetting);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, anisoSetting);
}
```

对 glewIsSupported()的调用测试显卡是否支持 AF。如果支持，我们将其设置为支持的最大采样程度，这个最大值使用 glGetFloatv()获取。然后使用 glTexParameterf()将其应用于激活纹理对象。结果如图 5.16 所示。请注意，图 5.11 中的大部分丢失细节已经恢复，同时仍然消除了闪烁的伪影。

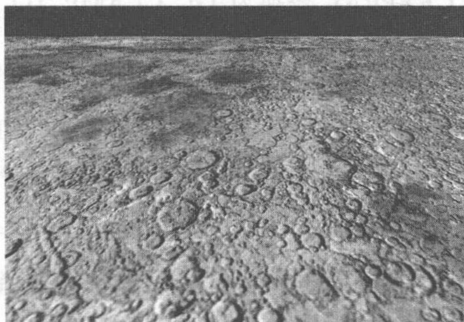


图 5.16 各向异性过滤

5.10 环绕和平铺

到目前为止，我们假设纹理坐标都落在[0...1]范围内。但是，OpenGL 实际上支持任何取值的纹理坐标。有几个选项可以用来指定当纹理坐标超出范围[0...1]时会发生什么。使用 glTexParameteri()设置所需的行为，选项如下。

- GL_REPEAT：忽略纹理坐标的整数部分，生成重复或“平铺”图案。这是默认行为。
- GL_MIRRORED_REPEAT：忽略整数部分，但是当整数部分为奇数时坐标反转，因此重复的图案在正常和镜像之间交替。
- GL_CLAMP_TO_EDGE：小于 0 或大于 1 的坐标分别设置为 0 和 1。
- GL_CLAMP_TO_BORDER：将[0...1]以外的纹素设置成指定的边框颜色。

例如，考虑一个金字塔，其纹理坐标已在[0...5]范围，而不是通常的[0...1]范围内定义。默认行为（GL_REPEAT），使用前面图 5.2 中显示的纹理图像，将导致纹理在表面上重复五次（有时称为“平铺”），如图 5.17 所示。

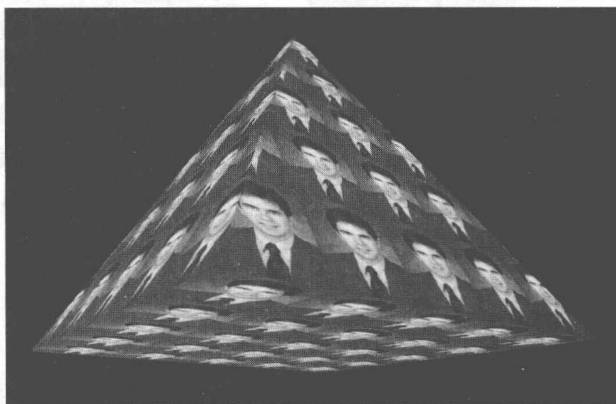


图 5.17 使用 GL_REPEAT 环绕的纹理坐标

为了使平铺块的外观在正常和镜像之间交替，我们可以指定以下内容：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

通过将 GL_MIRRORED_REPEAT 替换为 GL_CLAMP_TO_EDGE，可以指定将小于 0 或大于 1 的值分别设置为 0 和 1。

可以按如下方式来指定小于 0 或大于 1 的值输出“边框”颜色：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float redColor[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, redColor);
```

图 5.18（见彩插）中分别（从左到右）显示了每一个选项（镜像重复、夹紧到边缘和夹紧到边框）的效果，纹理坐标范围为−2～+3。

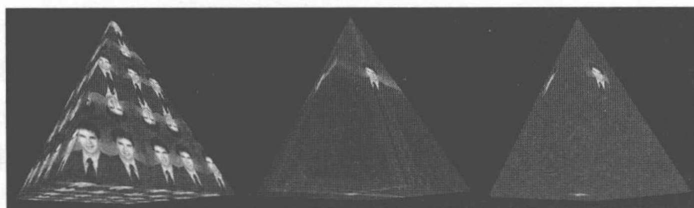


图 5.18 使用不同环绕选项的金字塔材质贴图

在中间的示例（夹紧到边缘）中，沿纹理图像边缘的像素向外复制。注意，作为副作用，金字塔面的左下和右下区域分别从纹理图像的左下和右下像素获得它们的颜色。

5.11 透视变形

我们已经看到，当纹理坐标从顶点着色器传递到片段着色器时，它们通过光栅着色器并被插值。我们还看到，这是自动线性插值的结果，总是在顶点属性上执行。

然而，在纹理坐标的情况下，线性插值可能导致具有透视投影的 3D 场景中的可以察觉的失真。

考虑一个由两个三角形组成的矩形，纹理贴图是棋盘图像，面向相机。当矩形围绕 *X* 轴旋转时，矩形的顶部会倾斜并远离相机，而矩形的下半部分则更靠近相机。因此，我们希望顶部的方块变小，底部的方块变大。但是，纹理坐标的线性插值将导致所有正方形的高度相等。沿着构成矩形的两个三角形之间的对角线的失真加剧。产生的失真如图 5.19 所示。

幸运的是，存在用于校正透视失真的算法，并且默认情况下，OpenGL 在光栅化期间会应用**透视校正算法**[OP14]。图 5.20 显示了由 OpenGL 正确呈现的相同的旋转棋盘。

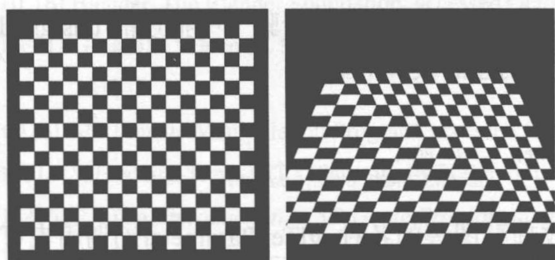


图 5.19 纹理透视失真

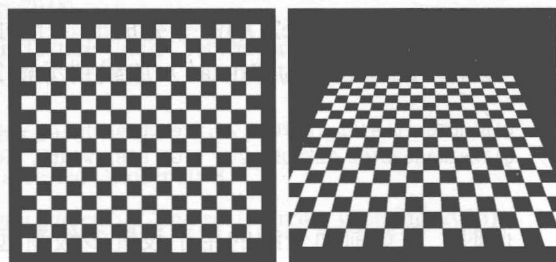


图 5.20 OpenGL 透视校正

虽然不常见，但可以通过在包含纹理坐标的顶点属性的声明中添加关键字“noperspective”来禁用 OpenGL 的透视校正。必须在顶点着色器和片段着色器中都这样添加。例如，顶点着色器中的顶点属性将声明如下：

```
noperspective out vec2 texCoord;
```

片段着色器中的相应属性声明：

```
noperspective in vec2 texCoord;
```

实际上，我使用了这种语法来生成图 5.19 中的扭曲棋盘格。

5.12 材质——更多 OpenGL 细节

我们在本书中使用的 SOIL2 纹理图像加载库具有使用起来相对简单和直观的优点。但是，在学习 OpenGL 时，使用 SOIL2 会产生一项我们不想要的后果，即用户会接触不到一些有用的重要 OpenGL 细节。在本节中，我们将描述程序员在没有纹理加载库（如 SOIL2）的情况下加载和使用纹理时需要了解的一些细节。

可以使用 C++和 OpenGL 函数直接将纹理图像文件数据加载到 OpenGL 中。虽然它有点复杂，但并不少见。一般步骤如下。

（1）使用 C++工具读取图像文件。

（2）生成 OpenGL 纹理对象。

（3）将图像文件数据复制到纹理对象中。

我们不会详细描述第一步——有太多方法了。在 opengl-tutorials.org（具体的教程页面为[OT18]）中很好地描述了一种方法，并使用 C++函数 fopen()和 fread()将数据从.bmp 图像文件读入 unsigned char 类型的数组中。

步骤 2 和步骤 3 更通用，主要涉及 OpenGL 调用。在第 2 步中，我们使用 OpenGL 的 glGenTextures()命令创建一个或多个纹理对象。例如，生成单个 OpenGL 纹理对象（使用整型引用 ID）可以按如下方式完成：

```
GLuint textureID;          // 或者 GLuint 类型的数组，如果需要创建多于一个纹理对象
glGenTextures(1, &textureID);
```

在步骤 3 中，我们将步骤 1 中的图像数据关联到步骤 2 中创建的纹理对象。这是使用 OpenGL 的 glTexImage2D()命令完成的。下面的示例将图像数据从步骤 1 中描述的 unsigned char 数组（此处表示为“data”）加载到步骤 2 中创建的纹理对象中：

```
glBindTexture(GL_TEXTURE_2D, textureID)
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, width, height, 0, GL_BGR,
                                                  GL_UNSIGNED_BYTE, data);
```

此时，本章前面介绍的用于设置多级渐远纹理贴图等的各种 glTexParameteri()调用也可以应用于纹理对象。我们现在也以与本章所述相同的方式使用整型引用（textureID）。

补充说明

研究人员开发了纹理单元的许多用途，不仅仅是场景中的纹理模型。在后面的章节中，我们将看到如何使用纹理单元来改变物体反射光线的方式，使其看起来凹凸不平。我们还可以使用纹理单元来存储“高度图”以生成地形，以及存储“阴影贴图”以有效地为场景添加阴影。这些用途将在后续章节中描述。

着色器还可以向纹理写入数据，允许着色器修改纹理图像，甚至将一个纹理的一部分复

制到另一个纹理的某个部分。

多级渐远纹理贴图和各向异性过滤不是减少纹理中的叠影伪影的唯一工具。例如，全屏抗锯齿（Full-scene anti-aliasing，FSAA）和其他超采样方法也可以改善 3D 场景中纹理的外观。虽然不是OpenGL核心的一部分，但它们通过OpenGL的扩展机制[OE16]在许多显卡上得到支持。

还有一种用于配置和管理纹理和采样器的替代机制。OpenGL 3.3 版引入了采样器对象（有时称为“采样器状态”——不要与采样器变量混淆），可用于保存一组独立于实际纹理对象的纹理设置。采样器对象附加到纹理单元，可以方便有效地更改纹理设置。本教材中显示的示例非常简单，我们决定暂不介绍采样器对象。对于感兴趣的读者，采样器对象的使用很容易学习，并且有许多优秀的在线教程（例如[GE11]）。

习题

5.1 如 5.11 节所述，通过在纹理坐标顶点属性中添加“noperspective”声明来修改程序 5.1。然后重新运行程序并将输出与原始输出进行比较。是否有任何明显的透视变形？

5.2 使用简单的“画图”程序（如 Windows“画图”或 GIMP[GI16]），绘制自己设计的手绘画面。然后使用您的图像在程序 5.1 中为金字塔添加纹理贴图。

5.3 （项目）修改程序 4.4，使“太阳”“行星”和“月亮”具有纹理。您可以继续使用已存在的形状，也可以使用任何您喜欢的纹理。通过搜索一些发布的代码示例可以获得立方体的纹理坐标，或者您可以手动构建它们（尽管这有点单调乏味）。

参考资料

[BL16] Blender, The Blender Foundation, accessed October 2018.

[GE11] Geeks3D, “OpenGL Sampler Objects: Control Your Texture Units,” September 8, 2011, accessed October 2018.

[GI16] GNU Image Manipulation Program, accessed October 2018.

[HT16] J. Hastings-Trew, JHT’s Planetary Pixel Emporium, accessed October 2018.

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[MA16] Maya, AutoDesk, Inc., accessed October 2018.

[OE16] OpenGL Registry, The Khronos Group, accessed July 2016.

[OP14] “OpenGL Graphics System: A Specification (version 4.4),” M. Segal and K. Akeley, March 19, 2014, accessed July 2016.

[OP16] OpenGL 4.5 Reference Pages, accessed July 2016.

[OT18] OpenGL Tutorial, “Loading BMP Images Yourself,” opengl-tutorial.org, accessed October 2018.

[SO17] Simple OpenGL Image Library 2 (SOIL2), *SpartanJ*, accessed October 2018.

[TU16] J. Turberville, Studio 522 Productions, Scottsdale, AZ.

[WI83] L. Williams, “Pyramidal Parametrics,” Computer Graphics 17, no. 3 (July 1983).

第 6 章　3D 模型

到目前为止，我们只处理了非常简单的 3D 对象，例如立方体和金字塔。这些对象非常简单，我们能够在源代码中明确列出所有顶点信息，并将其直接放入缓冲区。

然而，大多数有趣的 3D 场景包括的对象过于复杂，使得我们无法像之前那样继续手工构建它们。在本章中，我们将探索更复杂的对象模型，以及如何构建并将它们加载到场景中。

3D 建模本身就是一个广阔的领域，我们在这里讲到的必然非常有限。我们将重点关注以下两个主题：

- 通过程序来构建模型；
- 加载外部创建的模型。

虽然这只涉及丰富的 3D 建模领域中非常浅层的部分，但它将使我们能够在场景中包含各种复杂和逼真的细节对象。

6.1 程序构建模型——构建一个球体

某些类型的对象（例如球体、圆锥体等）具有数学定义，这些定义有助于算法生成。例如，对于半径为 R 的圆，围绕其圆周的点的坐标可以被很好地定义（见图 6.1）。

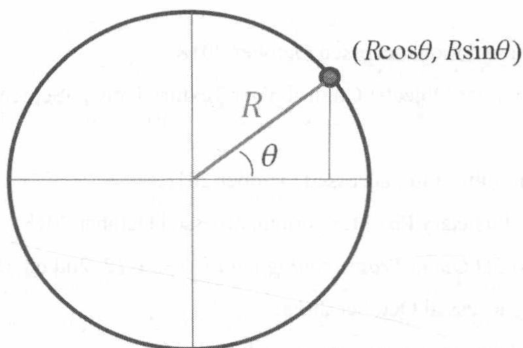


图 6.1　构成圆周的点

我们可以系统地使用圆的几何知识来通过算法建立球体模型。我们的策略如下。

（1）在整个球体上，选择表示一系列圆形“水平切片”的精度。见图 6.2 的左侧。

（2）将每个圆形切片的圆周细分为若干个点。见图 6.2 的右侧。更多的点和水平切片可以生成更精确、更平滑的球体模型。在我们的模型中，每个切片将具有相同数量的点。

（3）将顶点分组为三角形。一种方法是逐步遍历顶点，在每一步构建两个三角形。例如，当我们沿着图 6.3 中球体上 5 个彩色顶点这一行移动时，对于这 5 个顶点中的每一个，我们

构建了以相应颜色显示的两个三角形（见彩插，下面将更详细地描述这些步骤）。

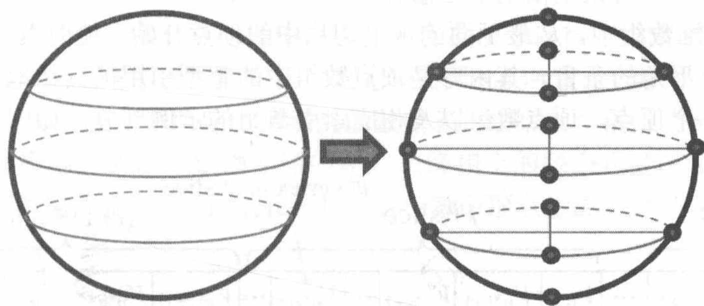


图 6.2 构建圆形顶点

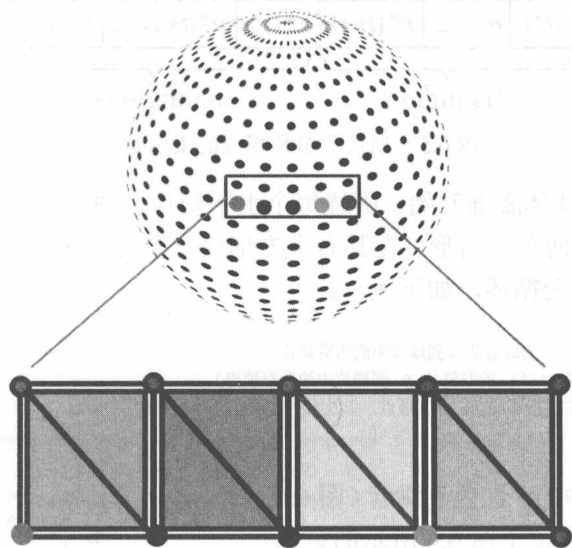


图 6.3 将顶点组合成三角形

（4）根据纹理图像的性质选择纹理坐标。在球体的情况下，存在许多地形纹理图像，假设我们选择这种纹理图像，想象一下，让这个图像围绕球体“包裹”，我们可以根据图像中纹素的最终对应位置为每个顶点指定纹理坐标。

（5）对于每个顶点，通常还希望生成法向量（Normal Vector）——垂直于模型表面的向量。我们将很快在第 7 章中将它们用于光照。

确定法向量可能很棘手，但是在球体的情况下，从球体中心指向顶点的向量恰好等于该顶点的法向量！图 6.4 说明了这个特点（球体的中心用“星形”表示）。

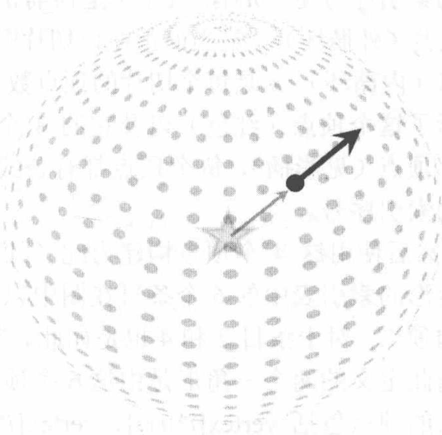


图 6.4 球体顶点法向量

一些模型使用索引定义三角形。请注意，在图 6.3 中，每个顶点出现在多个三角形中，这将导致每个顶点被多次指定。我们不希望这样做，而是会存储每个顶点一次，然后为三

角形的每个角指定索引，引用所需的顶点。我们需要存储每个顶点的位置、纹理坐标和法向量，因此这么做可以为大型模型节省内存。

顶点存储在一维数组中，从最下面的水平切片中的顶点开始。使用索引时，关联的索引数组包括每个三角形角的条目。其内容是顶点数组中的整型引用（具体地说，是下标）。假设每个切片包含 n 个顶点，顶点数组以及相应索引数组的示例部分，如图 6.5 所示。

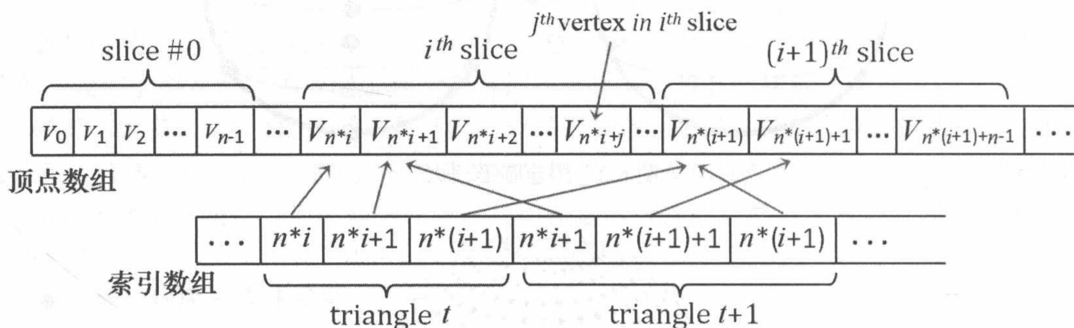


图 6.5 顶点数组和相应的索引数组

然后，我们可以从球体底部开始，围绕每个水平切片以圆形方式遍历顶点。当我们访问每个顶点时，我们构建两个三角形，在其右上方形成一个方形区域，如图 6.3 所示。我们将整个处理过程组织成嵌套循环，如下所示。

```
对于球体中的每个水平切片 i（i 的取值从 0 到球体中的所有切片）
{   对于切片 i 中的每个顶点 j（j 的取值从 0 到切片中的所有顶点）
    {   计算顶点 j 的指向右边相邻顶点、上方顶点，以及右上方顶点的两个三角形的索引
}   }
```

例如，考虑图 6.3 中的“红色”顶点（图 6.6 中重复出现）。这个顶点位于图 6.6 所示的黄色三角形的左下方，按照我们刚刚描述的循环，它的索引序号是 $i*n+j$，其中 i 是当前正在处理的切片（外循环），j 是当前正在该切片中处理的顶点（内循环），n 是每个切片的顶点数。图 6.6 显示了这个顶点（红色）以及它的 3 个相关的相邻顶点（见彩插），每个顶点都有公式显示它们的索引序号。

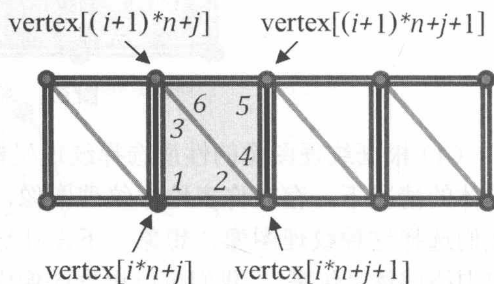


图 6.6 第 i 个切片中的第 j 个顶点的索引序号（n = 每个切片的顶点数）

然后使用这 4 个顶点构建为此（红色）顶点生成的两个三角形（以黄色显示）。这两个三角形的索引表中的 6 个条目在图中以数字 1～6 的顺序表示。注意，条目 3 和 6 都指向相同的顶点，对于条目 2 和 4 也是如此。当我们到达以红色突出显示的顶点（即 vertex[$i*n+j$]）时由此定义的两个三角形是由这 6 个顶点构成的——其中一个三角形的条目标记为 1、2、3，引用的顶点包括 vertex[$i*n+j$]、vertex[$i*n+j+1$]和 vertex[$(i+1)*n+j$]；另一个三角形的条目标记为 4、5、6，引用的顶点包括 vertex[$i*n+j+1$]、vertex[$(i+1)*n+j+1$]和 vertex[$(i+1)*n+j$]。

程序 6.1 显示了我们的球体模型的实现，类名为 Sphere。生成的球体的中心位于原点。这里还显示了使用 Sphere 的代码。请注意，每个顶点都存储在包含 GLM 类 vec2 和 vec3 实

例的 C++向量中（这与之前的示例不同，之前顶点存储在浮点数组中）。vec2 和 vec3 包括了获得所需的 *x*、*y* 和 *z* 分量浮点值的方法，然后就可以如前所述将它们放入浮点缓冲区。我们将这些值存储在可变长度 C++向量中，因为长度取决于运行时指定的切片数。

请注意 Sphere 类中三角形索引的计算，如前面的图 6.6 所述。变量“prec(precision)”指的是“精度”，在这里它被用来确定球形切片的数量和每个切片中的顶点数量。因为纹理贴图完全包裹在球体周围，所以在纹理贴图的左右边缘相交的每个点处需要一个额外的重合顶点。因此，顶点的总数是(prec+1)*(prec+1)。由于每个顶点生成 6 个三角形索引，因此索引的总数是 prec*prec*6。

程序 6.1 程序生成的球体

```
球体类（Sphere.cpp）

#include <cmath>
#include <vector>
#include <iostream>
#include <glm\glm.hpp>
#include "Sphere.h"
using namespace std;

Sphere::Sphere() {
  init(48);
}

Sphere::Sphere(int prec) {    // prec 是精度，也就是切片的数量
  init(prec);
}

float Sphere::toRadians(float degrees) { return (degrees * 2.0f * 3.14159f) / 360.0f; }

void Sphere::init(int prec) {
  numVertices = (prec + 1) * (prec + 1);
  numIndices = prec * prec * 6;
  // std::vector::push_back() 在向量的末尾增加一个新元素，并为向量长度加 1
  for (int i = 0; i < numVertices; i++) { vertices.push_back(glm::vec3()); }
  for (int i = 0; i < numVertices; i++) { texCoords.push_back(glm::vec2()); }
  for (int i = 0; i < numVertices; i++) { normals.push_back(glm::vec3()); }
  for (int i = 0; i < numIndices; i++) { indices.push_back(0); }

  // 计算三角形顶点
  for (int i = 0; i <= prec; i++) {
      for (int j = 0; j <= prec; j++) {
          float y = (float)cos(toRadians(180.0f - i * 180.0f / prec));
          float x = -(float)cos(toRadians(j*360.0f / prec)) * (float)abs(cos(asin(y)));
          float z = (float)sin(toRadians(j*360.0f / prec)) * (float)abs(cos(asin(y)));
          vertices[i*(prec + 1) + j] = glm::vec3(x, y, z);
          texCoords[i*(prec + 1) + j] = glm::vec2(((float)j / prec), ((float)i / prec));
          normals[i*(prec + 1) + j] = glm::vec3(x,y,z);
      }
  }

  // 计算三角形索引
  for (int i = 0; i<prec; i++) {
      for (int j = 0; j<prec; j++) {
          indices[6 * (i*prec + j) + 0] = i*(prec + 1) + j;
          indices[6 * (i*prec + j) + 1] = i*(prec + 1) + j + 1;
```

```
            indices[6 * (i*prec + j) + 2] = (i + 1)*(prec + 1) + j;
            indices[6 * (i*prec + j) + 3] = i*(prec + 1) + j + 1;
            indices[6 * (i*prec + j) + 4] = (i + 1)*(prec + 1) + j + 1;
            indices[6 * (i*prec + j) + 5] = (i + 1)*(prec + 1) + j;
        }
    }
}
// 读取函数
int Sphere::getNumVertices() { return numVertices; }
int Sphere::getNumIndices() { return numIndices; }
std::vector<int> Sphere::getIndices() { return indices; }
std::vector<glm::vec3> Sphere::getVertices() { return vertices; }
std::vector<glm::vec2> Sphere::getTexCoords() { return texCoords; }
std::vector<glm::vec3> Sphere::getNormals() { return normals; }
```

球体头文件（Sphere.h）

```
#include <cmath>
#include <vector>
#include <glm\glm.hpp>

class Sphere
{
private:
   int numVertices;
   int numIndices;
   std::vector<int> indices;
   std::vector<glm::vec3> vertices;
   std::vector<glm::vec2> texCoords;
   std::vector<glm::vec3> normals;
   void init(int);
   float toRadians(float degrees);

public:
   Sphere(int prec);
   int getNumVertices();
   int getNumIndices();
   std::vector<int> getIndices();
   std::vector<glm::vec3> getVertices();
   std::vector<glm::vec2> getTexCoords();
   std::vector<glm::vec3> getNormals();
};
```

使用球体类

```
. . .
#include "Sphere.h"
. . .
Sphere mySphere(48);
. . .
void setupVertices(void) {
   std::vector<int> ind = mySphere.getIndices();
   std::vector<glm::vec3> vert = mySphere.getVertices();
   std::vector<glm::vec2> tex = mySphere.getTexCoords();
   std::vector<glm::vec3> norm = mySphere.getNormals();

   std::vector<float> pvalues;     // 顶点位置
   std::vector<float> tvalues;     // 纹理坐标
   std::vector<float> nvalues;     // 法向量
```

```
    int numIndices = mySphere.getNumIndices();
    for (int i = 0; i < numIndices; i++) {
        pvalues.push_back((vert[ind[i]]).x);
        pvalues.push_back((vert[ind[i]]).y);
        pvalues.push_back((vert[ind[i]]).z);

        tvalues.push_back((tex[ind[i]]).s);
        tvalues.push_back((tex[ind[i]]).t);

        nvalues.push_back((norm[ind[i]]).x);
        nvalues.push_back((norm[ind[i]]).y);
        nvalues.push_back((norm[ind[i]]).z);
    }

    glGenVertexArrays(1, vao);
    glBindVertexArray(vao[0]);
    glGenBuffers(3, vbo);

    // 把顶点放入缓冲区 #0
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, pvalues.size()*4, &pvalues[0], GL_STATIC_DRAW);

    // 把纹理坐标放入缓冲区 #1
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glBufferData(GL_ARRAY_BUFFER, tvalues.size()*4, &tvalues[0], GL_STATIC_DRAW);

    // 把法向量放入缓冲区 #2
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glBufferData(GL_ARRAY_BUFFER, nvalues.size()*4, &nvalues[0], GL_STATIC_DRAW);
}
在display()中
. . .
glDrawArrays(GL_TRIANGLES, 0, mySphere.getNumIndices());
. . .
```

使用 Sphere 类时，每个顶点位置和法向量需要 3 个值，但每个纹理坐标只需要两个值。这反映在 Sphere.h 文件中显示的向量（vertices、texCoords 和 normals）的声明中，稍后数据从这些向量中加载到缓冲区中。

值得注意的是，虽然在构建球体的过程中使用了索引，但存储在 VBO 中的最终球体顶点数据不使用索引。相反，当 setupVertices()循环遍历球体索引时，它会在 VBO 中为每个索引条目生成单独的（通常是冗余的）顶点条目。OpenGL 确实有一种索引顶点数据的机制；为简单起见，我们在此示例中没有使用它，但我们将在下一个示例中使用 OpenGL 的索引。

从几何形状到现实世界的物体，使用程序的方式可以创建许多其他的模型。其中最著名的一个是“犹他茶壶”[CH16]，在 1975 年由马丁·纽厄尔（Martin Newell）开发，使用各种贝塞尔曲线和曲面。OpenGL Utility Toolkit（或“GLUT”）[GL16]甚至包括了绘制茶壶的程序（见图 6.7）。我们在本书中没有涉及 GLUT，但贝塞尔曲面将在第 11 章中介绍。

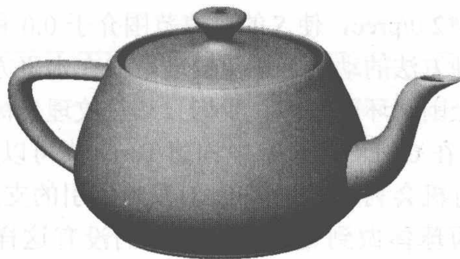


图 6.7 OpenGL GLUT 茶壶

6.2 OpenGL 索引——构建一个环面

6.2.1 环面

用于产生环面的算法可以在各种网站上找到。Paul Baker 逐步描述了定义圆形切片，然后围绕圆圈旋转切片以形成环面的方法[PP07]。图 6.8 显示了侧面和上面的两种视图。

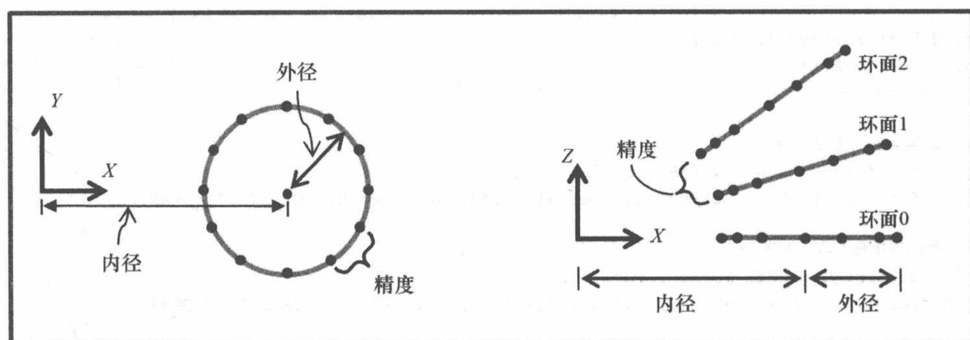


图 6.8 构建一个环面

生成环面顶点位置的方式与构建球体的方式有很大不同。对于环面，算法将一个顶点定位到原点的右侧，然后在 *XY* 平面上的圆中让这个顶点围绕 *Z* 轴旋转，以形成“环”。然后，将这个环“向外”移动“内半径”那么长的距离。在构建这些顶点时，为每个顶点计算纹理坐标和法向量。还会额外为每个顶点生成与环面表面相切的向量（称为**切向量**）。

围绕 *Y* 轴旋转最初的这个环，形成用来构成环面的其他环的顶点。通过围绕 *Y* 轴旋转最初的环的切向量和法向量来计算每个结果顶点的切向量和法向量。在顶点创建之后，逐个环地遍历所有顶点，并且对于每个顶点生成两个三角形。两个三角形的六个索引表条目的生成方式和之前的球体类似。

我们为剩余的环选择纹理坐标的策略，是将它们排列成使得纹理图像的 *S* 轴环绕环面的水平周边的一半，然后再对另一半重复。当我们绕 *Y* 轴旋转生成环时，我们指定一个从 1 开始并增加到指定精度的变量环（再次称为“prec”）。然后我们将 *S* 纹理坐标值设置为 ring*2.0/prec，使 *S* 的取值范围介于 0.0 和 2.0 之间，然后每当纹理坐标大于 1.0 时减去 1.0。这种方法的动机是避免纹理图像在水平方向上过度“拉伸”。反之，如果我们确实希望纹理完全围绕环面拉伸，我们只须从纹理坐标计算中删除“*2.0”乘数即可。

在 C++/OpenGL 中构建 Torus 类可以用与 Sphere 类几乎完全相同的方式完成。但是，我们有机会利用 OpenGL 对顶点索引的支持来利用我们在构建环面时创建的索引（我们也可以为球体做到这一点，但我们没有这样做）。对于具有数千个顶点的超大型模型，使用 OpenGL 索引可以提高性能，因此我们将描述如何执行此操作。

6.2.2 OpenGL 中的索引

在我们的球体和环面模型中，我们生成一个引用顶点数组的整型索引数组。在球体的情况下，我们使用索引列表来构建一组完整的单个顶点，并将它们加载到 VBO 中，就像我们在前面章节的示例中所做的那样。实例化环面并将其顶点、法向量等加载到缓冲区中可以采用与程序 6.1 中类似的方式完成，但我们将使用 OpenGL 的索引。

使用 OpenGL 索引时，我们还需要将索引本身加载到 VBO 中。我们生成一个额外的 VBO 用于保存索引。由于每个索引值只是一个整型引用，我们首先将索引数组复制到整型的 C++ 向量中，然后使用 glBufferData()将向量加载到新增的 VBO 中，指定 VBO 的类型为 GL_ELEMENT_ARRAY_BUFFER（这会告诉 OpenGL 这个 VBO 包含索引）。执行此操作的代码可以添加到 setupVertices():

```
std::vector<int> ind = myTorus.getIndices();      // 环面索引的读取函数返回整型向量类型的索引
. . .
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);    // vbo #3 是新增的 VBO
glBufferData(GL_ELEMENT_ARRAY_BUFFER, ind.size()*4, &ind[0], GL_STATIC_DRAW);
```

在 display()方法中，我们将 glDrawArrays()调用替换为 glDrawElements()调用，它告诉 OpenGL 利用索引 VBO 来查找要绘制的顶点。我们还使用 glBindBuffer()启用包含索引的 VBO，指定哪个 VBO 包含索引并且是 GL_ELEMENT_ARRAY_BUFFER 类型。代码如下：

```
numTorusIndices = myTorus.getNumIndices();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
```

有趣的是，即使我们在 C++/OpenGL 应用程序中进行了更改，实现了索引，用于绘制球体的着色器对于环面来说仍然可以继续工作，不需要修改。OpenGL 能够识别 GL_ELEMENT_ARRAY_BUFFER 的存在并利用它来访问顶点属性。

程序 6.2 显示了一个基于 Baker 实现的名为 Torus 的类。“内”和“外”变量指的是图 6.9 中相应的内半径和外半径。prec 变量具有与球体类似的作用，对顶点数量和索引数量进行类似的计算。相比之下，确定法向量比使用球体复杂得多。我们使用了 Baker 描述中给出的策略，其中计算了两个切向量（Baker 称为 sTangent 和 tTangent，尽管通常称为“切向量（tangent）”和“副切向量（bitangent）”），它们的叉乘积形成法向量。

在本书的其余部分中，我们将在许多示例中使用此环面类（以及前面描述的球体类）。

程序 6.2 程序生成的环面

```
Torus 类 (Torus.cpp)

#include <cmath>
#include <vector>
#include <iostream>
#include "Torus.h"
using namespace std;

Torus::Torus() {
   prec = 48;
   inner = 0.5f;
```

```
        outer = 0.2f;
        init();
    }

    Torus::Torus(float innerRadius, float outerRadius, int precIn) {
        prec = precIn;
        inner = innerRadius;
        outer = outerRadius;
        init();
    }

    float Torus::toRadians(float degrees) { return (degrees * 2.0f * 3.14159f) / 360.0f; }

    void Torus::init() {
        numVertices = (prec + 1) * (prec + 1);
        numIndices = prec * prec * 6;
        for (int i = 0; i < numVertices; i++) { vertices.push_back(glm::vec3()); }
        for (int i = 0; i < numVertices; i++) { texCoords.push_back(glm::vec2()); }
        for (int i = 0; i < numVertices; i++) { normals.push_back(glm::vec3()); }
        for (int i = 0; i < numVertices; i++) { sTangents.push_back(glm::vec3()); }
        for (int i = 0; i < numVertices; i++) { tTangents.push_back(glm::vec3()); }
        for (int i = 0; i < numIndices; i++) { indices.push_back(0); }

        // 计算第一个环
        for (int i = 0; i < prec + 1; i++) {
            float amt = toRadians(i*360.0f / prec);
            // 绕原点旋转点，形成环，然后将它们向外移动
            glm::mat4 rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 0.0f, 1.0f));
            glm::vec3 initPos(rMat * glm::vec4(outer, 0.0f, 0.0f, 1.0f));
            vertices[i] = glm::vec3(initPos + glm::vec3(inner, 0.0f, 0.0f));

            // 为环上的每个顶点计算纹理坐标
            texCoords[i] = glm::vec2(0.0f, ((float)i / (float)prec));

            // 计算切向量和法向量，第一个切向量是绕 Z 轴旋转的 Y 轴
            rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 0.0f, 1.0f));
            tTangents[i] = glm::vec3(rMat * glm::vec4(0.0f, -1.0f, 0.0f, 1.0f));
            sTangents[i] = glm::vec3(glm::vec3(0.0f, 0.0f, -1.0f));    // 第二个切向量是 -Z 轴
            normals[i] = glm::cross(tTangents[i], sTangents[i]);     // 它们的叉乘积就是法向量
        }

        // 绕 Y 轴旋转最初的那个环，形成其他的环
        for (int ring = 1; ring < prec + 1; ring++) {
            for (int vert = 0; vert < prec + 1; vert++) {

            // 绕 Y 轴旋转最初那个环的顶点坐标
            float amt = (float)( toRadians(ring * 360.0f / prec));
            glm::mat4 rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));
            vertices[ring*(prec + 1) + i] = glm::vec3(rMat * glm::vec4(vertices[i], 1.0f));

            // 计算新环顶点的纹理坐标
            texCoords[ring*(prec + 1) + vert] = glm::vec2((float)ring*2.0f / (float)prec, texCoords
                [vert].t);
            if (texCoords[ring*(prec + 1) + i].s > 1.0) texCoords[ring*(prec+1)+i].s -= 1.0f;

            // 绕 Y 轴旋转切向量和副切向量
            rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));
            sTangents[ring*(prec + 1) + i] = glm::vec3(rMat * glm::vec4(sTangents[i], 1.0f));
              rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));
              tTangents[ring*(prec + 1) + i] = glm::vec3(rMat * glm::vec4(tTangents[i], 1.0f));
```

```
            // 绕 Y 轴旋转法向量
            rMat = glm::rotate(glm::mat4(1.0f), amt, glm::vec3(0.0f, 1.0f, 0.0f));
            normals[ring*(prec + 1) + i] = glm::vec3(rMat * glm::vec4(normals[i], 1.0f));
    } }

    // 按照逐个顶点的两个三角形，计算三角形索引
    for (int ring = 0; ring < prec; ring++) {
        for (int vert = 0; vert < prec; vert++) {
            indices[((ring*prec + vert) * 2) * 3 + 0] = ring*(prec + 1) + vert;
            indices[((ring*prec + vert) * 2) * 3 + 1] = (ring + 1)*(prec + 1) + vert;
            indices[((ring*prec + vert) * 2) * 3 + 2] = ring*(prec + 1) + vert + 1;
            indices[((ring*prec + vert) * 2 + 1) * 3 + 0] = ring*(prec + 1) + vert + 1;
            indices[((ring*prec + vert) * 2 + 1) * 3 + 1] = (ring + 1)*(prec + 1) + vert;
            indices[((ring*prec + vert) * 2 + 1) * 3 + 2] = (ring + 1)*(prec + 1) + vert + 1;
} } }

// 环面索引和顶点的访问函数
int Torus::getNumVertices() { return numVertices; }
int Torus::getNumIndices() { return numIndices; }
std::vector<int> Torus::getIndices() { return indices; }
std::vector<glm::vec3> Torus::getVertices() { return vertices; }
std::vector<glm::vec2> Torus::getTexCoords() { return texCoords; }
std::vector<glm::vec3> Torus::getNormals() { return normals; }
std::vector<glm::vec3> Torus::getStangents() { return sTangents; }
std::vector<glm::vec3> Torus::getTtangents() { return tTangents; }
```

环面头文件（Torus.h）

```
#include <cmath>
#include <vector>
#include <glm\glm.hpp>
class Torus
{
private:
    int numVertices;
    int numIndices;
    int prec;
    float inner;
    float outer;
    std::vector<int> indices;
    std::vector<glm::vec3> vertices;
    std::vector<glm::vec2> texCoords;
    std::vector<glm::vec3> normals;
    std::vector<glm::vec3> sTangents;
    std::vector<glm::vec3> tTangents;
    void init();
    float toRadians(float degrees);

public:
    Torus();
    Torus(float innerRadius, float outerRadius, int prec);
    int getNumVertices();
    int getNumIndices();
    std::vector<int> getIndices();
    std::vector<glm::vec3> getVertices();
    std::vector<glm::vec2> getTexCoords();
    std::vector<glm::vec3> getNormals();
    std::vector<glm::vec3> getStangents();
    std::vector<glm::vec3> getTtangents();
```

```
};
```

使用 Torus 类（用 OpenGL 索引）

```
. . .
#include "Torus.h"
. . .
Torus myTorus(0.5f, 0.2f, 48);
. . .
void setupVertices(void) {
   std::vector<int> ind = myTorus.getIndices();
   std::vector<glm::vec3> vert = myTorus.getVertices();
   std::vector<glm::vec2> tex = myTorus.getTexCoords();
   std::vector<glm::vec3> norm = myTorus.getNormals();

   std::vector<float> pvalues;
   std::vector<float> tvalues;
   std::vector<float> nvalues;

   int numVertices = myTorus.getNumVertices();
   for (int i = 0; i < numVertices; i++) {
        pvalues.push_back(vert[i].x);
        pvalues.push_back(vert[i].y);
        pvalues.push_back(vert[i].z);

        tvalues.push_back(tex[i].s);
        tvalues.push_back(tex[i].t);

        nvalues.push_back(norm[i].x);
        nvalues.push_back(norm[i].y);
        nvalues.push_back(norm[i].z);
   }
   glGenVertexArrays(1, vao);
   glBindVertexArray(vao[0]);
   glGenBuffers(4, vbo);           // 像以前一样生成 VBO，并新增一个用于索引

   glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);           // 顶点位置
   glBufferData(GL_ARRAY_BUFFER, pvalues.size() * 4, &pvalues[0], GL_STATIC_DRAW);

   glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);           // 纹理坐标
   glBufferData(GL_ARRAY_BUFFER, tvalues.size() * 4, &tvalues[0], GL_STATIC_DRAW);

   glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);           // 法向量
   glBufferData(GL_ARRAY_BUFFER, nvalues.size() * 4, &nvalues[0], GL_STATIC_DRAW);

   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);  // 索引
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, ind.size() * 4, &ind[0], GL_STATIC_DRAW);
}
```

在 display()中

```
. . .
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
glDrawElements(GL_TRIANGLES, myTorus.getNumIndices(), GL_UNSIGNED_INT, 0);
```

请注意，在使用 Torus 类的代码中，setupVertices()中的循环现在只存储一次与每个顶点关联的数据，而不是每个索引条目存储一次（如球体示例中的情况）。这种差异也反映在要输入 VBO 的数据的数组声明大小中。另请注意，在环面示例中，不是在检索顶点数据时使

用索引值，而是直接将它们简单地加载到 VBO #3 中。由于此 VBO 被指定为 GL_ELEMENT_ARRAY_BUFFER，OpenGL 知道该 VBO 包含顶点索引。

图 6.9 显示了实例化环面并使用砖纹理对其进行纹理化的结果。

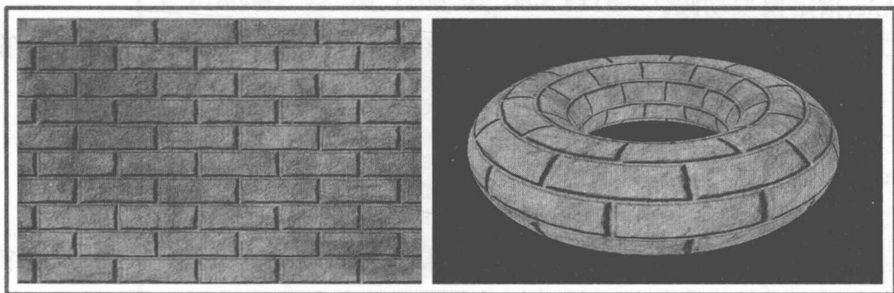


图 6.9　程序生成的环面

6.3　加载外部构建的模型

复杂的 3D 模型，例如在视频游戏或计算机生成的电影中的人物角色，通常使用建模工具生成。这种“DCC”（数字内容创建）工具使人们（例如艺术家）能够在 3D 空间中构建任意形状并自动生成顶点、纹理坐标、顶点法向量等。有太多这样的工具，此处无法一一列出，有几个例子是 MAYA、Blender、Lightwave、Cinema4D 等。Blender 是免费和开源的。图 6.10 显示了编辑 3D 模型时的 Blender 屏幕示例。

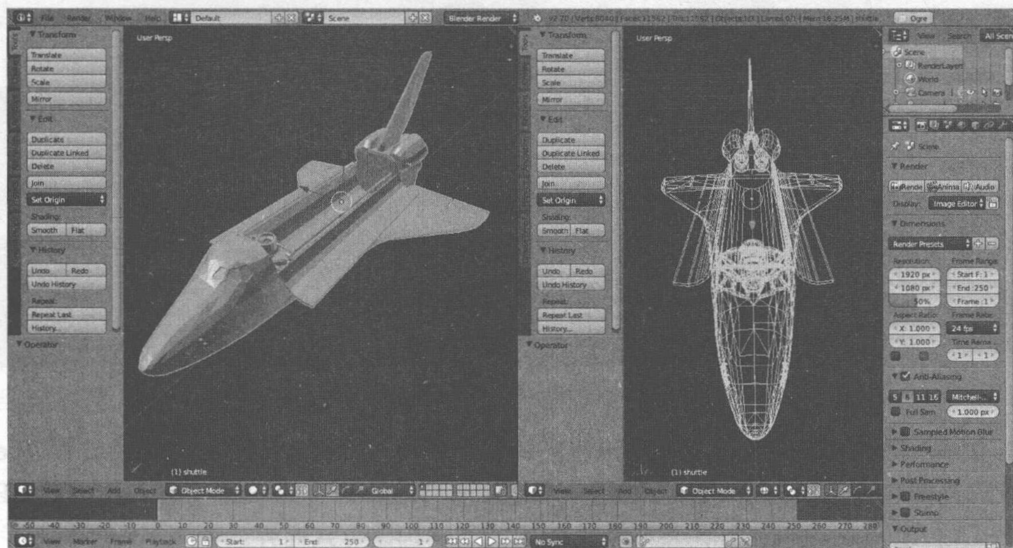


图 6.10　Blender 模型创建示例[BL16]

为了让我们在 OpenGL 场景中使用 DCC 工具创建的模型，需要以我们可以读取（导入）到我们程序中的格式保存（导出）该模型。有好几种标准的 3D 模型文件格式；再次说明，有太多无法一一列出，有一些例子是 Wavefront（.obj）、3D Studio Max（.3ds）、斯坦福扫描存储库（.ply）、Ogre3D（.mesh），供参考。其中最简单的是 Wavefront（通常被称为 OBJ），

所以我们将仔细讲解它。

OBJ 文件很简单，我们可以相对容易地开发一个基本的导入器。在 OBJ 文件中，通过文本行的形式指定顶点几何数据、纹理坐标、法向量和其他信息。它有一些限制——例如，OBJ 文件无法指定模型动画。

OBJ 文件中的行，以字符标记开头，表示该行上的数据类型。一些常见的标签包括：

- v-几何（顶点位置）数据；
- vt-纹理坐标；
- vn-顶点法向量；
- f-面（通常是三角形中的顶点）。

还有其他标签可以用来存储对象名称、使用的材质、曲线、阴影和许多其他细节。我们这里只讨论上面列出的 4 个标签，这些标签足以导入各种复杂模型。

假设我们使用 Blender 构建一个简单的金字塔，例如我们为程序 4.3 开发的金字塔。图 6.11 是在 Blender 中创建的类似的金字塔的屏幕截图。

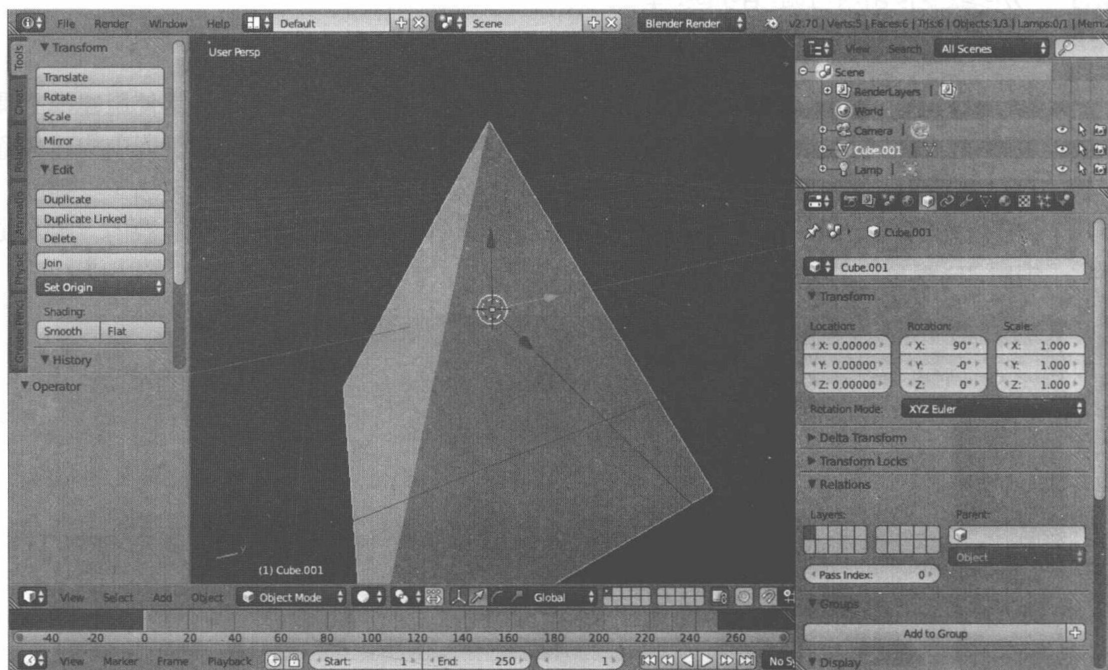


图 6.11　在 Blender 中构建的金字塔

在 Blender 中，如果我们现在导出我们的金字塔模型，指定.obj 格式，并设置 Blender 输出纹理坐标和顶点法向量，则会创建一个包含所有这些信息的 OBJ 文件。生成的 OBJ 文件如图 6.12 所示。（纹理坐标的实际值可能因模型的构建方式而异。）

我们对 OBJ 文件的重要部分进行了颜色标记以供参考。顶部以“#”开头的行是由 Blender 放置的注释，我们的导入器忽略了这些注释。接下来是以“o”开头的行，给出对象的名称。我们的导入器也可以忽略这一行。之后，有一行以“s”开头，指定表面不应该被平滑。我们的代码也会忽略以“s”开头的行。

OBJ 文件中的第一部分有实际内容的行是以“v”开头的那些（第 4 行～第 8 行）。它们

指定金字塔模型的 5 个顶点相对于原点的 *X*、*Y* 和 *Z* 局部空间坐标。在这里，原点位于金字塔的中心。

```
# Blender v2.70 (sub 0) OBJ File: ''
# www.blender.org
o Pyramid
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 0.000000 1.000000 0.000000
vt 0.515829 0.258220
vt 0.515829 0.750612
vt 0.023438 0.750612
vt 0.370823 0.790246
vt 0.820312 0.388210
vt 0.820312 0.991264
vt 0.566135 0.988689
vt 0.015625 0.742493
vt 0.566135 0.496298
vt 0.015625 0.250102
vt 0.566135 0.003906
vt 1.000000 0.000000
vt 1.000000 0.603054
vt 0.550510 0.402036
vt 0.023438 0.258220
vn 0.000000 -1.000000 0.000000
vn 0.894427 0.447214 0.000000
vn -0.000000 0.447214 0.894427
vn -0.894427 0.447214 -0.000000
vn 0.000000 0.447214 -0.894427
s off
f 2/1/1 3/2/1 4/3/1
f 1/4/2 5/5/2 2/6/2
f 2/7/3 5/8/3 3/9/3
f 3/9/4 5/10/4 4/11/4
f 5/12/5 1/13/5 4/14/5
f 1/15/1 2/1/1 4/3/1
```

图 6.12　金字塔导出的 OBJ 文件

红色的值（以“vt”开头）是各种纹理坐标。纹理坐标列表比顶点列表长的原因是一些顶点参与多个三角形，并且在这些情况下可能使用不同的纹理坐标。

绿色的值（以“vn”开头）是各种法向量。该列表通常也比顶点列表长（尽管在该示例中不是这样），同样是因为一些顶点参与多个三角形，并且在那些情况下可能使用不同的法向量。

在文件底部附近标记为紫色的值（以“f”开头）指定三角形（即“面”）。在此示例中，每个面（三角形）具有 3 个元素，每个元素具有由“/”分隔的 3 个值（OBJ 也允许其他格式）。每个元素的值分别是顶点列表、纹理坐标和法向量的索引。例如，第三个面是：

f 2/7/3　5/8/3　3/9/3

这表明顶点列表中的第 2 个、第 5 个和第 3 个顶点（蓝色）组成了一个三角形（请注意 OBJ 索引从 1 开始）。相应的纹理坐标是红色部分中纹理坐标列表中的第 7 项、第 8 项和第 9

项。所有 3 个顶点都具有相同的法向量，也就是以绿色显示的法向量列表中的第 3 项。

OBJ 格式的模型并不要求具有法向量，甚至纹理坐标。如果模型没有纹理坐标或法向量，则面的数值将仅指定顶点索引：

f 2 5 3

如果模型具有纹理坐标，但不具有法向量，则格式如下：

f 2/7 5/8 3/9

并且，如果模型具有法向量但没有纹理坐标，则格式为：

f 2//3 5//3 3//3

模型具有数万个顶点并不罕见。对于所有可以想象的应用场景，几乎都可以在互联网上下载到数百种这样的模型，包括动物、建筑物、汽车、飞机、神话生物、人等。

在互联网上可以获得可以导入 OBJ 模型的复杂程序各不相同的导入程序。编写一个非常简单的 OBJ 加载器函数也并不困难，它可以处理我们看到的基本标记（v、vt、vn 和 f）。程序 6.3 显示了一个这样的加载器，尽管功能非常有限。它包含一个类来保存任意的导入模型，该模型又调用导入器。

在我们讲述简单 OBJ 导入器的代码之前，我们必须警告读者其局限性。

- 它仅支持包含所有 3 个面属性字段的模型。也就是说，顶点位置、纹理坐标和法向量都必须以 f #/#/# #/#/# #/#/#这种形式存在。
- 材质标签将被忽略——必须使用第 5 章中描述的方法完成纹理化。
- 仅支持由单个三角形网格组成的 OBJ 模型（OBJ 格式支持复合模型，但我们的简单导入器不支持）。
- 它假设每行上的元素只用一个空格分隔。

如果您的 OBJ 模型不满足上述所有条件，并且您希望使用程序 6.3 中的简单加载程序导入它，则可能仍然可行。通常可以将这样的模型加载到 Blender 中，然后将其导出到另一个满足加载器限制条件的 OBJ 文件中。例如，如果模型不包含法向量，则可以让 Blender 在导出修改后的 OBJ 文件时生成法向量。

我们的 OBJ 加载器的另一个限制与索引有关。在前面的描述中提到了“f”标签允许混合和匹配顶点位置、纹理坐标和法向量的可能性。例如，两个不同的“面”行可以包括指向相同 v 条目但是不同 vt 条目的索引。遗憾的是，OpenGL 的索引机制不支持这种灵活性——OpenGL 中的索引条目只能指向特定的顶点及其属性。这使得在某种程度上编写 OBJ 模型加载器变得复杂，因为我们不能简单地将三角形面条目中的引用复制到索引数组中。相反，使用 OpenGL 索引需要确保面条目的 v、vt 和 vn 值的整个组合在索引数组中都有自己的引用。一种更简单但效率更低的替代方案是为每个三角形面条目创建一个新顶点。尽管使用 OpenGL 索引具有节省空间的优势（特别是在加载较大模型时），但为了清晰，我们选择这种更简单的方法。

ModelImporter 类包含一个 parseOBJ()函数，它逐行读取 OBJ 文件，分别处理 v、vt、vn 和 f 这 4 种情况。在每种情况下，提取行上的后续数字，首先使用 erase()跳过初始的 v、vt、vn 或 f 字符，然后使用 C++ stringstream 类的“>>”运算符提取每个后续参数值，然后将它们存储在 C++浮点向量中。当处理面（f）条目时，使用 C++浮点向量中的对应条目构建顶

点，包括顶点位置、纹理坐标和法向量。

ModelImporter 类和 ImportedModel 类包含在同一个文件中，ImportedModel 类通过将导入的顶点放入 vec2 和 vec3 对象的向量中，简化了加载和访问 OBJ 文件顶点的过程。回想一下这些 GLM 类；我们在这里使用它们来存储顶点位置、纹理坐标和法向量。然后，ImportedModel 类中的读取函数使它们可用于 C++/OpenGL 应用程序，其方式与 Sphere 和 Torus 类中的方式相同。

在 ModelImporter 和 ImportedModel 类之后是一系列调用示例，加载 OBJ 文件，然后将顶点信息传输到一组 VBO 中以供后续渲染。

图 6.13 显示了从 NASA 网站[NA16]下载的 OBJ 格式的航天飞机渲染模型，使用程序 6.3 中的代码导入，并使用程序 5.1 中的代码和相应的带有各向异性过滤的 NASA 纹理图像文件进行纹理化。该纹理图像是使用 UV 映射的示例，其中模型中的纹理坐标被仔细地映射到纹理图像的特定区域。（如第 5 章所述，UV 映射的细节超出了本书的范围。）

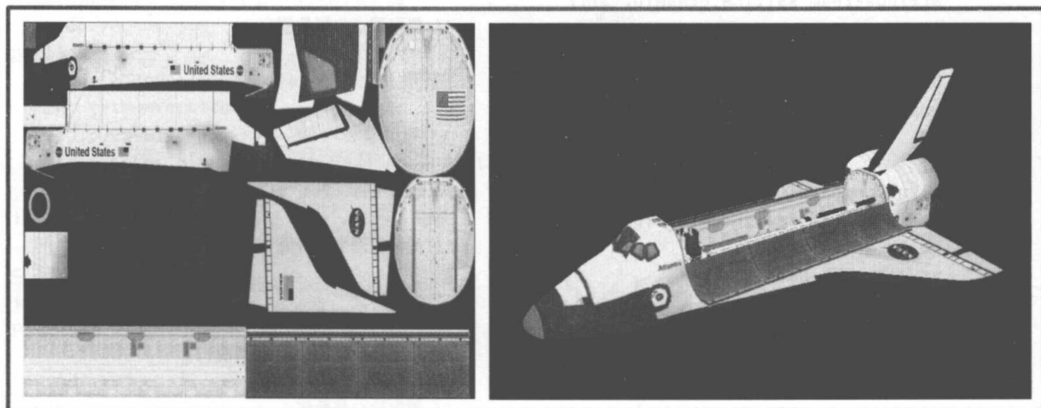


图 6.13 带有纹理的 NASA 航天飞机模型

程序 6.3 简化的（有限制的）OBJ 加载器

```
ImportedModel 和 ModelImporter 类(ImportedModel.cpp)

#include <fstream>
#include <sstream>
#include <glm\glm.hpp>
#include "ImportedModel.h"
using namespace std;

// ------------ ImportedModel 类

ImportedModel::ImportedModel(const char *filePath) {
  ModelImporter modelImporter = ModelImporter();
  modelImporter.parseOBJ(filePath);            // 使用 modelImporter 获取顶点信息
  numVertices = modelImporter.getNumVertices();
  std::vector<float> verts = modelImporter.getVertices();
  std::vector<float> tcs = modelImporter.getTextureCoordinates();
  std::vector<float> normals = modelImporter.getNormals();

  for (int i = 0; i < numVertices; i++) {
      vertices.push_back(glm::vec3(verts[i*3], verts[i*3+1], verts[i*3+2]));
      texCoords.push_back(glm::vec2(tcs[i*2], tcs[i*2+1]));
      normalVecs.push_back(glm::vec3(normals[i*3], normals[i*3+1], normals[i*3+2]));
```

```
} }

int ImportedModel::getNumVertices() { return numVertices; }        // accessors
std::vector<glm::vec3> ImportedModel::getVertices() { return vertices; }
std::vector<glm::vec2> ImportedModel::getTextureCoords() { return texCoords; }
std::vector<glm::vec3> ImportedModel::getNormals() { return normalVecs; }

// -------------- ModelImporter类

ModelImporter::ModelImporter() {}

void ModelImporter::parseOBJ(const char *filePath) {
   float x, y, z;
   string content;
   ifstream fileStream(filePath, ios::in);
   string line = "";
   while (!fileStream.eof()) {
      getline(fileStream, line);
      if (line.compare(0, 2, "v ") == 0) {                   // 顶点位置("v"的情况)
         stringstream ss(line.erase(0, 1));
         ss >> x; ss >> y; ss >> z;                          // 提取顶点位置数值
         vertVals.push_back(x);
         vertVals.push_back(y);
         vertVals.push_back(z);
      }

      if (line.compare(0, 2, "vt") == 0) {                   // 纹理坐标("vt"的情况)
         stringstream ss(line.erase(0, 2));
         ss >> x; ss >> y;                                   // 提取纹理坐标数值
         stVals.push_back(x);
         stVals.push_back(y);
      }
      if (line.compare(0, 2, "vn") == 0) {                   // 顶点法向量("vn"的情况)
         stringstream ss(line.erase(0, 2));
         ss >> x; ss >> y; ss >> z;                          // 提取法向量数值
         normVals.push_back(x);
         normVals.push_back(y);
         normVals.push_back(z);
      }
      if (line.compare(0, 2, "f") == 0) {                    // 三角形面("f"的情况)
         string oneCorner, v, t, n;
         stringstream ss(line.erase(0, 2));
         for (int i = 0; i < 3; i++) {
            getline(ss, oneCorner, ' ');                     // 提取三角形面引用
            stringstream oneCornerSS(oneCorner);
            getline(oneCornerSS, v, '/');
            getline(oneCornerSS, t, '/');
            getline(oneCornerSS, n, '/');

            int vertRef = (stoi(v) - 1) * 3;                 // "stoi"将字符串转化为整型
            int tcRef = (stoi(t) - 1) * 2;
            int normRef = (stoi(n) - 1) * 3;

            triangleVerts.push_back(vertVals[vertRef]);          // 构建顶点向量
            triangleVerts.push_back(vertVals[vertRef + 1]);
            triangleVerts.push_back(vertVals[vertRef + 2]);

            textureCoords.push_back(stVals[tcRef]);              // 构建纹理坐标向量
            textureCoords.push_back(stVals[tcRef + 1]);
```

```
            normals.push_back(normVals[normRef]);          // 法向量的向量
            normals.push_back(normVals[normRef + 1]);
            normals.push_back(normVals[normRef + 2]);
} } } }

int ModelImporter::getNumVertices() { return (triangleVerts.size()/3); }     // 读取函数
std::vector<float> ModelImporter::getVertices() { return triangleVerts; }
std::vector<float> ModelImporter::getTextureCoordinates() { return textureCoords; }
std::vector<float> ModelImporter::getNormals() { return normals; }
```

ImportedModel 和 ModelImporter 头文件（ImportedModel.h）

```
#include <vector>

class ImportedModel
{
private:
   int numVertices;
   std::vector<glm::vec3> vertices;
   std::vector<glm::vec2> texCoords;
   std::vector<glm::vec3> normalVecs;
public:
   ImportedModel(const char *filePath);
   int getNumVertices();
   std::vector<glm::vec3> getVertices();
   std::vector<glm::vec2> getTextureCoords();
   std::vector<glm::vec3> getNormals();
};

class ModelImporter
{
private:
   // 从 OBJ 文件读取的数值
   std::vector<float> vertVals;
   std::vector<float> stVals;
   std::vector<float> normVals;

   // 保存为顶点属性以供后续使用的数值
   std::vector<float> triangleVerts;
   std::vector<float> textureCoords;
   std::vector<float> normals;

public:
   ModelImporter();
   void parseOBJ(const char *filePath);
   int getNumVertices();
   std::vector<float> getVertices();
   std::vector<float> getTextureCoordinates();
   std::vector<float> getNormals();
};
```

使用模型导入器

```
. . .
ImportedModel myModel("shuttle.obj");      // 在顶层声明中
. . .

void setupVertices(void) {
   std::vector<glm::vec3> vert = myModel.getVertices();
   std::vector<glm::vec2> tex = myModel.getTextureCoords();
```

```
    std::vector<glm::vec3> norm = myModel.getNormals();
    int numObjVertices = myModel.getNumVertices();

    std::vector<float> pvalues;        // 顶点位置
    std::vector<float> tvalues;        // 纹理坐标
    std::vector<float> nvalues;        // 法向量

    for (int i = 0; i < numObjVertices(); i++) {
       pvalues.push_back((vert[i]).x);
       pvalues.push_back((vert[i]).y);
       pvalues.push_back((vert[i]).z);
       tvalues.push_back((tex[i]).s);
       tvalues.push_back((tex[i]).t);
       nvalues.push_back((norm[i]).x);
       nvalues.push_back((norm[i]).y);
       nvalues.push_back((norm[i]).z);
    }

    glGenVertexArrays(1, vao);
    glBindVertexArray(vao[0]);
    glGenBuffers(numVBOs, vbo);

    // 顶点位置的 VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, pvalues.size() * 4, &pvalues[0], GL_STATIC_DRAW);

    // 纹理坐标的 VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glBufferData(GL_ARRAY_BUFFER, tvalues.size() * 4, &tvalues[0], GL_STATIC_DRAW);

    // 法向量的 VBO
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glBufferData(GL_ARRAY_BUFFER, nvalues.size() * 4, &nvalues[0], GL_STATIC_DRAW);
}
```

在 display()中

```
. . .
glDrawArrays(GL_TRIANGLES, 0, myModel.getNumVertices());
```

补充说明

虽然我们讨论了使用 DCC 工具创建 3D 模型，但我们没有讨论如何使用这些工具。相关教程超出了本书的范围，但是所有流行的工具都有大量的教程视频材料文档可供使用，例如 Blender 和 MAYA。

3D 建模的主题本身就是一个丰富的研究领域。我们在本章中所说只是一个基本的介绍，重点是它与 OpenGL 的关系。许多大学提供 3D 建模的全部课程，并且我们也鼓励有兴趣学习更多的读者参考一些提供更多细节的流行资源（例如：[BL16, CH11, VA12]）。

我们重申，本章中介绍的 OBJ 导入器是很有限的，并且只能处理 OBJ 格式支持的一部分功能。虽然足以满足我们的需求，但它会在某些 OBJ 文件上失败。在这些情况下，有必要首先将模型加载到 Blender（或 MAYA 等）工具中，然后将其重新导出为符合导入器限制的 OBJ 文件，如本章前面所述。

习题

6.1 修改程序 4.4，使“太阳”“行星”和“月亮”成为纹理球体，如图 4.11 所示。

6.2 （项目）修改您的习题 6.1 的程序，以使得图 6.16 中导入的 NASA 航天飞机对象也绕“太阳”运行。您需要试验应用于航天飞机的尺度和旋转方式，使其看起来更逼真。

6.3 （研究和项目）了解如何使用 Blender 创建自己的 3D 对象的基础知识。想要在您的 OpenGL 应用程序中充分利用 Blender，您将需要学习如何使用 Blender 的 UV 展开工具来生成纹理坐标和相关的纹理图像。然后，您可以将对象导出为 OBJ 文件，并使用程序 6.3 中的代码加载它。

参考资料

[BL16] Blender, The Blender Foundation, accessed October 2018.

[CH11] A. Chopine, *3D Art Essentials: The Fundamentals of 3D Modeling, Texturing, and Animation* (Focal Press, 2011).

[CH16] Computer History Museum, accessed October 2018.

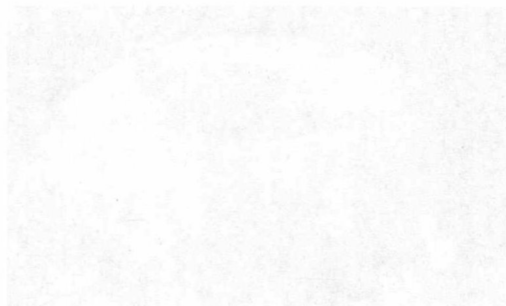
[GL16] GLUT and OpenGL Utility Libraries, accessed October 2018.

[NA16] NASA 3D Resources, accessed October 2018.

[PP07] P. Baker, *Paul's Projects*, 2007, accessed October 2018.

[VA12] V. Vaughan, *Digital Modeling* (New Riders, 2012).

[VE16] Visible Earth, NASA Goddard Space Flight Center Image, accessed October 2018.



第 7 章　光照

光照以不同的方式影响着我们世界的外观，有时甚至是很戏剧化的方式。当手电筒照射在物体上时，我们会期望它面向光线的一侧看起来更亮。我们所居住的地球，在中午朝向太阳时候被照得很亮，但随着它的自转，同一个地点的亮度会逐渐由白天转变为傍晚，直到午夜变得完全黑暗。物体对光的反射也各不相同。物体除了颜色的差别，也可以具有不同的反射特性。考虑两个物体，在都是绿色的情况下，其中一个是布制的，而另一个是抛光钢材质的——那么后者看起来会更“闪亮”。

7.1　光照模型

我们所观察到的光是高能量源发出的光子，经过反射直到一些光子到达我们的眼睛的产物。不幸的是，在计算上模拟这个自然过程是不可行的，因为这需要模拟并跟踪大量光子的运动，即向我们的场景添加海量的对象（和矩阵）。因此，我们需要的是光照模型。

光照模型（Lighting model）有时也被称为着色模型（Shading model），在着色器编程存在的情况下，这可能有点令人困惑。有时又使用术语反射模型（Reflection model），进一步使术语复杂化。我们将尽力坚持使用简单而实用的术语。

现在最常见的光照模型称为“ADS”模型，因为它们基于标记为 A、D 和 S 的 3 种类型的反射。

- 环境光反射（Ambient reflection）模拟低级光照，影响场景中的所有物体。
- 漫反射（Diffuse reflection）根据光线的入射角度调整物体亮度。
- 镜面反射（Specular reflection）用以展示物体的光泽，通过在物体表面上，光线最直接地反射到我们的眼睛的位置，策略性地放置适当大小的高光来实现。

ADS 模型可用于模拟不同的光照效果和各种材质。

图 7.1（见彩插）展示了位置光对于闪亮黄金环面的环境光反射、漫反射和镜面反射分量。

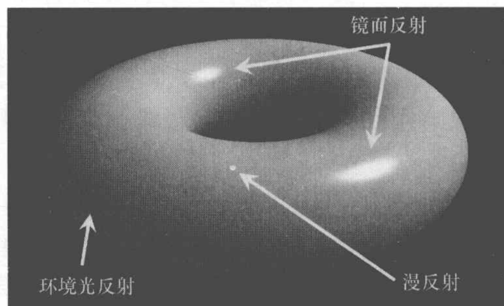


图 7.1　ADS 光照分量

回想一下，场景的绘制最终是由片段着色器为屏幕上的每个像素输出颜色而实现的。使用 ADS 光照模型需要指定由于像素的 RGBA 输出值上的光照而产生的分量。因素包括：

- 光源类型及其环境、漫反射和镜面反射特性；
- 对象材质的环境、漫反射和镜面反射特征；

- 对象的材质指定为“光泽”；
- 光线照射物体的角度；
- 从中查看场景的角度。

7.2 光源

光源有许多类型，每种光源具有不同的特性，需要不同的步骤来模拟其效果。常见光源类型有：

- 全局光（通常称为“全局环境光”，因为它仅包含环境光组件）；
- 定向光（或“远距离光”）；
- 位置光（或“点光源”）；
- 聚光灯。

全局环境光是最简单的光源模型。它没有光源位置——无论场景中的对象在何处，其上的每个像素都有着相同的光照。全球环境光照模拟了现实世界中的一种光线现象，即光线经过很多次反射，其光源和方向都已经无法确定。全局环境光仅具有环境光反射分量，用 RGBA 值设定；它没有漫反射或镜面反射分量。例如，全局环境光可以定义如下：

```
float globalAmbient[4] = { 0.6f, 0.6f, 0.6f, 1.0f };
```

RGBA 的取值范围为 0～1，全局环境光通常被建模为偏暗的白光，其中 RGB 各值设为 0～1 的相同的小数，alpha 设置为 1。

定向光或远距离光也没有源位置，但它具有方向。它可以用来模拟光源距离非常远，以至于光线接近平行的情况，例如阳光。通常在这种情况下，我们可能只对建模光照感兴趣，而对发光的物体不感兴趣。定向光对物体的影响取决于光照角度，物体在朝向定向光的一侧比在切向或相对侧更亮。建模定向光需要指定其方向（以向量形式）及其环境、漫反射和镜面特征（以 RGBA 值）。指向 Z 轴负方向的红色定向光可以指定如下：

```
float dirLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f };
float dirLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float dirLightSpecular[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float dirLightDirection[3] = { 0.0f, 0.0f, -1.0f };
```

在已经有全局环境光的情况下，定向光的环境光分量看起来似乎是多余的。然而，当光源“开启”或“关闭”时，全局环境光和定向光的环境光分量的区别就很明显了。当“开启”时，总环境光分量将如预期的那样增加。上面的例子中，我们只使用了很小的环境光分量。在实际场景中，应当根据场景的需要平衡两个环境光分量。

位置光在 3D 场景中具有特定位置。靠近场景的光源，例如台灯，蜡烛等。像定向光一样，位置光的效果取决于撞击角度；但是，它没有方向，因为它对场景中的每个顶点的光照方向都不同。位置光还可以包含衰减因子，以模拟它们的强度随距离减小的程度。与我们看到的其他类型的光源一样，位置光具有指定为 RGBA 值的环境光反射、漫反射和镜面反射特性。位置（5,2,−3）处的红色位置光可以指定如下例：

```
float posLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f };
float posLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float posLightSpecular[4] = { 1.0f,0.0f, 0.0f, 1.0f };
float posLightLocation[3] = { 5.0f, 2.0f, -3.0f };
```

衰减因子有多种建模方式。其中一种方式是使用恒定、线性和二次方（分别称为 k_c, k_l 和 k_q）衰减，并引入非负可调参数。这些参数与离光源的距离（d）结合进行计算：

$$attenuationFactor = \frac{1}{k_c + k_l d + k_q d^2}$$

将这个因子与光的强度相乘可以使距光更远时，光的强度衰减更多。注意，k_c应当永远设置为大于等于 1 的值，从而使得衰减因子落入[0…1]区间，并当 d 增大时接近于 0。

聚光灯（spotlight）同时具有位置和方向。其“锥形”效果可以使用 0°～90°的截光角 θ 来模拟，指定光束的半宽度，并使用衰减指数来模拟随光束角度的强度变化。如图 7.2 所示，我们确定聚光灯方向与从聚光灯到像素的向量之间的角度 φ。当 φ 小于 θ 时，我们通过将 φ 的余弦提高到衰减指数来计算强度因子（当 φ 大于 θ 时，强度因子设置为 0）。结果是强度因子的范围为 0～1。衰减指数会影响当角度 φ 增加时，强度因子趋于 0 的速率。然后将强度因子乘以光的强度以模拟锥形效果。

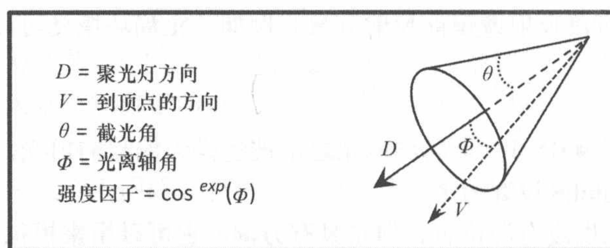


图 7.2 聚光灯参数

位于（5,2,–3）向下照射 Z 轴负方向的红色聚光灯可以表示为：

```
float spotLightAmbient[4] = { 0.1f, 0.0f, 0.0f, 1.0f };
float spotLightDiffuse[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
float spotLightSpecular[4] = { 1.0f,0.0f, 0.0f, 1.0f };
float spotLightLocation[3] = { 5.0f, 2.0f, -3.0f };
float spotLightDirection[3] = { 0.0f, 0.0f, -1.0f };
float spotLightCutoff = 20.0f;
float spotLightExponent = 10.0f;
```

聚光灯也可以引入衰减因子。我们没有在上面的代码中展示它们，不过，聚光灯衰减因子可以用与前述定向光源相同的方式实现。历史上，自 1986 年皮克斯的著名动画《小台灯》（*Luxo Jr.*）出现起，聚光灯就成为了计算机图形学的标志。

当设计拥有许多光源的系统时，程序员应该考虑创建相应的类结构，如定义 Light 类以及其子类 GlobalAmbient、Directional、Positional 以及 Spotlight。由于聚光灯同时具有定向光和位置光的特性，这里就值得使用 C++的多继承能力，让 Spotlight 类同时继承于实现位置光和定向光的类。在示例中，由于内容足够简单，因此我们在当前版本中没有加入这种层次结构。

7.3　材质

我们场景中物体的“外观”目前仅使用颜色和纹理进行表现。增加的光照使得我们可以加入表面的反射特性。即对象如何与我们的 ADS 光照模型相互作用。这可以通过将每个对象视为“由某种材质制成”来建模。

通过指定 4 个值（我们已经熟悉其中 3 个值——环境光、漫反射和镜面 RGB 颜色），可以在 ADS 光照模型中模拟材质。第四种叫作光泽，正如我们将要看到的那样，它被用来为所选材质建立一个合适的镜面高光。目前许多不同类型的常见材质已经有 ADS 和光泽度值了。例如，“锡镴”可以指定如下：

```
float pewterMatAmbient[4] = { .11f, .06f, .11f, 1.0f };
float pewterMatDiffuse[4] = { .43f, .47f, .54f, 1.0f };
float pewterMatSpecular[4] = { .33f, .33f, .52f, 1.0f };
float pewterMatShininess = 9.85f;
```

一些其他材质的 ADS RGBA 值见图 7.3（引自[BA16]）。

有时候一些其他特性也属于材质特性。透明度由 RGBA 标准中的第四个（alpha）通道的不透明度来实现。取值为 1.0 是表示完全不透明，取值为 0 时表示完全透明。对于大多数材质而言，只需要把不透明度设置为 1.0 就行了，但是对于某些特定的材质，加入一些透明度是很重要的。例如，图 7.3 中材质“玉”和“珍珠”都含有少量透明度（取值略微小于 1.0）以显得更加真实。

放射性有时也包含在 ADS 材质规范中。在模拟自身发光的材质（例如磷光材质）时非常有用。

材质	环境光RGBA 漫反射RGBA 反射RGBA	光泽
黄金	0.24725, 0.1995, 0.0745, 1.0 0.75164, 0.60648, 0.22648, 1.0 0.62828, 0.5558, 0.36607, 1.0	51.2
玉	0.135, 0.2225, 0.1575, 0.95 0.54, 0.89, 0.63, 0.95 0.3162, 0.3162, 0.3162, 0.95	12.8
珍珠	0.25, 0.20725, 0.20725, 0.922 1.00, 0.829, 0.829, 0.922 0.2966, 0.2966, 0.2966, 0.922	11.264
银	0.19225, 0.19225, 0.19225, 1.0 0.50754, 0.50754, 0.50754, 1.0 0.50827, 0.50827, 0.50827, 1.0	51.2

图 7.3　其他材质的 ADS 系数

没有纹理的物体在渲染时，通常需要指定材质特性。因此，预定义一些可供选择的材质，在使用时会很方便。因此我们需要在 Utils.cpp 文件中添加如下代码：

```
// 黄金材质 — 环境光、漫反射、镜面反射和光泽
float * Utils::goldAmbient() { static float a[4] = { 0.2473f, 0.1995f, 0.0745f, 1 }; return
    (float * ) a; }
float * Utils::goldDiffuse() { static float a[4] = { 0.7516f, 0.6065f, 0.2265f, 1 }; return
    (float * ) a; }
float * Utils::goldSpecular() { static float a[4] = { 0.6283f, 0.5559f, 0.3661f, 1 }; return
    (float * ) a; }
float Utils::goldShininess() { return 51.2f; }

// 白银材质 — 环境光、漫反射、镜面反射和光泽
```

```
float * Utils::silverAmbient() { static float a[4] = { 0.1923f, 0.1923f, 0.1923f, 1 }; return
    (float * ) a; }
float * Utils::silverDiffuse() { static float a[4] = { 0.5075f, 0.5075f, 0.5075f, 1 }; return
    (float * ) a; }
float * Utils::silverSpecular() { static float a[4] = { 0.5083f, 0.5083f, 0.5083f, 1 }; return
    (float * ) a; }
float Utils::silverShininess() { return 51.2f; }

// 青铜材质 — 环境光、漫反射、镜面反射和光泽
float * Utils::bronzeAmbient() { static float a[4] = { 0.2125f, 0.1275f, 0.0540f, 1 }; return
    (float * ) a; }
float * Utils::bronzeDiffuse() { static float a[4] = { 0.7140f, 0.4284f, 0.1814f, 1 }; return
    (float * ) a; }
float * Utils::bronzeSpecular() { static float a[4] = { 0.3936f, 0.2719f, 0.1667f, 1 }; return
    (float * ) a; }
float Utils::bronzeShininess() { return 25.6f; }
```

这样在 init()函数中或全局中为物体指定“黄金”材质就非常容易了，如下所示。

```
float* matAmbient = Utils::goldAmbient();
float* matDiffuse = Util::goldDiffuse();
float* matSpecular = util.goldSpecular();
float matShininess = util.goldShininess();
```

注意，目前为止的各小节中，我们所用来实现的光照和材质特性的代码并没有引入光照。这些代码仅仅提供了用于描述并存储场景中元素所需光照和材质特性的一种方式。我们仍然需要自己计算光照。编写计算光照的代码需要在我们的着色器代码中引入一些严肃的数学过程。因此，让我们先来看看在 C++/OpenGL 和 GLSL 图形程序中实现 ADS 光照的基础。

7.4 ADS 光照计算

当我们绘制场景时，每个顶点坐标都会进行变换以将 3D 世界模拟到 2D 屏幕上。每个像素的颜色都是光栅化、纹理贴图以及插值的结果。现在我们需要加入一个新的步骤来调整这些光栅化之后的像素颜色，以便反应场景中的光照和材质。我们需要做的基础 ADS 计算是确定每个像素的反射强度（Reflection Intensity，I）。计算过程如下：

$$I_{\text{observed}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

我们需要计算每个光源对于每个像素的环境光反射、漫反射和镜面反射分量，并求和。当然，这些计算都基于场景内的光源类型以及渲染中模型的材质类型。

环境光分量是最简单的。它的值是场景环境光与材质环境光分量的乘积：

$$I_{\text{ambient}} = Light_{\text{ambient}} * Material_{\text{ambient}}$$

请记住光与材质亮度都是 RGB 值，计算可以更准确地描述为：

$$I_{\text{ambient}}^{\text{red}} = Light_{\text{ambient}}^{\text{red}} * Material_{\text{ambient}}^{\text{red}}$$
$$I_{\text{ambient}}^{\text{green}} = Light_{\text{ambient}}^{\text{green}} * Material_{\text{ambient}}^{\text{green}}$$
$$I_{\text{ambient}}^{\text{blue}} = Light_{\text{ambient}}^{\text{blue}} * Material_{\text{ambient}}^{\text{blue}}$$

漫反射分量会更复杂一些，因为它基于光对于平面的入射角。朗伯余弦定律（1760 年出版）确定了表面反射的光量与光入射角的余弦成正比。可以建模为如下公式：

$$I_{\text{diffuse}} = Light_{\text{diffuse}} * Material_{\text{diffuse}} * \cos(\theta)$$

与上面的计算相同，实际计算中所用到的是红、绿、蓝分量。

确定入射角 θ 需要（a）求解从所绘制向量到光源的向量（或者与光照方向相反的向量），（b）求解所渲染物体表面的法（垂直）向量。让我们将其分别称为 $\boldsymbol{L}$ 和 $\boldsymbol{N}$，如图 7.4 所示。

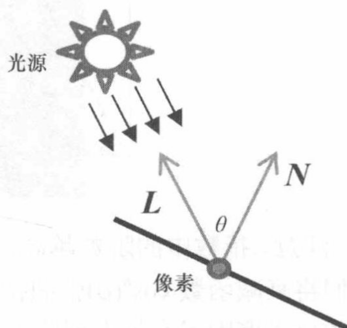


图 7.4　入射角

基于场景中光的物理特性，向量 $\boldsymbol{L}$ 可以通过对光照方向向量取反，或通过计算像素位置到光源位置的向量得到。计算向量 $\boldsymbol{N}$ 会麻烦一些——法向量有可能已经在模型中给出了，但是如果模型没有给出法向量 $\boldsymbol{N}$，那么就需要基于周围顶点位置，在几何上对向量 $\boldsymbol{N}$ 进行估计。在本章剩下的内容中，我们假设所渲染的模型每个顶点都包含法向量（使用建模工具如 MAYA 或 Blender 创建的模型，通常都包含法向量）。

事实上，在计算法向量时，没必要计算出 θ 角本身的角度。我们真正需要的是 $\cos(\theta)$。在第 3 章中讲过，这可以通过点乘计算得出。因此，漫反射分量可以通过如下公式得出：

$$I_{\text{diffuse}} = Light_{\text{diffuse}} * Material_{\text{diffuse}} * (\hat{N} \bullet \hat{L})$$

漫反射分量仅当表面暴露在光照中时起作用，即当 $-90 < \theta < 90$，$\cos(\theta) > 0$ 时。因此，我们需要将之前等式的最右项替换为：

$$\max((\hat{N} \bullet \hat{L}), 0)$$

镜面反射分量决定所渲染的像素是否需要作为“镜面高光”的一部分变亮。它不止与光源的入射角相关，也与光在表面上的反射角以及观察点与反光表面之间的夹角相关。

在图 7.5 中，$\boldsymbol{R}$ 代表光反射的方向，$\boldsymbol{V}$（叫作观察向量 view vector）是从像素到眼睛的向量。注意，$\boldsymbol{V}$ 是对从眼睛到像素的向量取反（在相机空间中，眼睛位于原点）。在 $\boldsymbol{R}$ 与 $\boldsymbol{V}$ 之间的小夹角 φ 越小，眼睛越靠近光轴，或者说看向反射光，因此像素的镜面高光分量也就越大（像素看来应该更亮）。

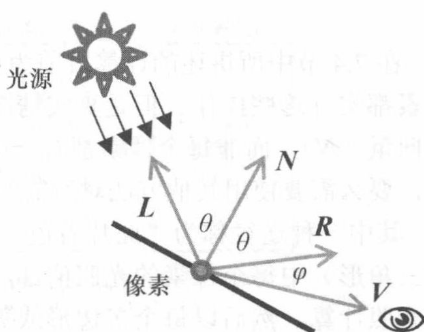


图 7.5　观察点入射角

φ 用于计算镜面反射分量的方式取决于所渲染物体的“光泽度”。极端闪亮的物体，如镜子，其镜面高光非常小——它们将入射的光直接反射给了眼睛。不那么闪亮的物体，其镜面高光会扩散开来，因此高光会包含更多的像素。

反光度通常用衰减函数来建模，这个衰减函数用来表达随着角度 φ 的增大，镜面反射分量降低到 0 的速度。我们可以用 $\cos(\varphi)$ 来对衰减进行建模，通过余弦函数的乘方来增减反光度，如 $\cos(\varphi), \cos^2(\varphi), \cos^3(\varphi), \cos^{10}(\varphi), \cos^{50}(\varphi)$ 等，如图 7.6 所示。

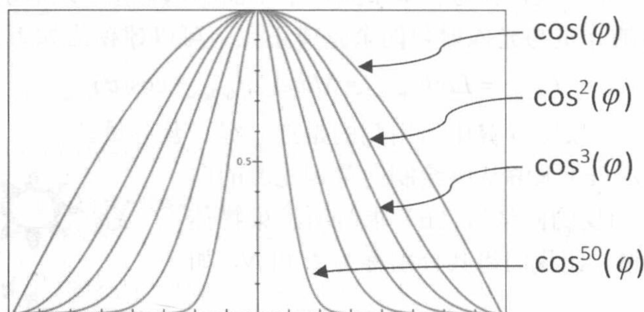


图 7.6 以余弦指数建模的反光度

注意，指数中的阶数越高，衰减越快，因此在视角光轴外的反光像素镜面反射分量越小。我们将衰减函数 $\cos^n(\varphi)$中的指数 n 叫作材质的反光度因子。注意在之前的图 7.3 中，每个材质的反光度因子在最右列给出。

现在我们可以给出完整的镜面反射计算：

$$I_{\text{spec}} = Light_{\text{spec}} * Material_{\text{spec}} * \max(0, (\hat{\boldsymbol{R}} \bullet \hat{\boldsymbol{V}})^n)$$

注意，与之前计算漫反射一样，我们使用了 max()函数。在本例中，我们需要确保镜面反射分量不使用 $\cos(\varphi)$所产生的负值，如果使用了负值，则会有奇怪的伪影，如“暗”镜面高光。

同时，如之前一样，真正的计算中包含了红、绿、蓝 3 个分量。

7.5 实现 ADS 光照

在 7.4 节中所讲述的计算目前为止都是理论上的，其中包含的假设是，我们可以对每个像素都实行这些操作。但是真实情况会更复杂，通常模型中只有用来定义模型的顶点才有法向量（$\boldsymbol{N}$），而非每个像素都有。因此我们要么需要计算每个像素的法向量，这会非常耗时，要么需要使用其他方法对所需的值进行估计，以实现足够好的效果。

其中一种途径称为“面片着色”或“平坦着色”。这里我们假定所渲染图元（如多边形或三角形）中每个像素的光照值都一样。因此我们只需要对模型每个多边形的一个顶点进行光照计算，然后以每个多边形或每个三角形为基础，将计算结果的光照值复制到相邻的像素中。

现在面片着色几乎已经不再使用，因为其渲染结果看来不够真实，同时现代硬件已经可以进行更加精确的计算了。图 7.7 中展示了一个面片着色环面的例子，其中每个三角形都作为平坦的反射表面。

虽然某些情况下，面片着色可能已经够用了（或者故意使用其效果），但是通常“平滑着色”是一种更好的途径。在平滑着色的过程中，会对每个像素计算光照强度。现代显卡的并行处理功能，以及 OpenGL 图形管线中的插值渲染让平滑着色变得可行。

我们将会观察两个流行的平滑着色方法：Gouraud 着色和 Phong 着色。

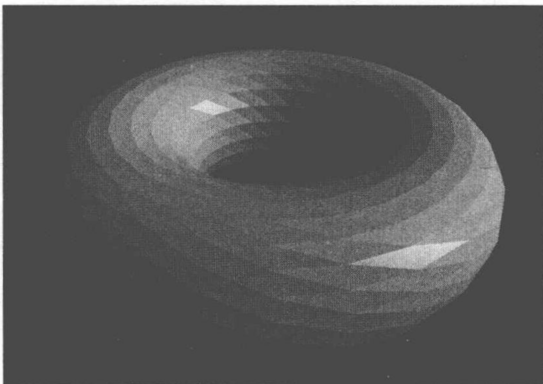


图 7.7 面片着色的环面

7.5.1 Gouraud 着色（双线性光强插值法）

法国计算机科学家 Henri Gouraud 在 1971 年发表的平滑着色算法后来被称为 Gouraud 着色[GO71]。由于使用了 3D 图形管线（如 OpenGL）中的自动插值渲染，它特别适用于现代显卡。Gouraud 着色过程如下。

（1）确定每个顶点的颜色，以及光照相关计算。

（2）允许正常的光栅化过程在插入像素时对颜色也进行插值（同时也对光照进行插值）。

在 OpenGL 中，这表示大多数光照计算都是在顶点着色器中完成的，片段着色器仅做传递并展示自动插值的光照后的颜色。

图 7.8 展示了在场景中包含环面和单一位置光的情况下，我们将会用来在 OpenGL 中实现 Gouraud 着色器的策略。程序 7.1 中实现了这个策略。

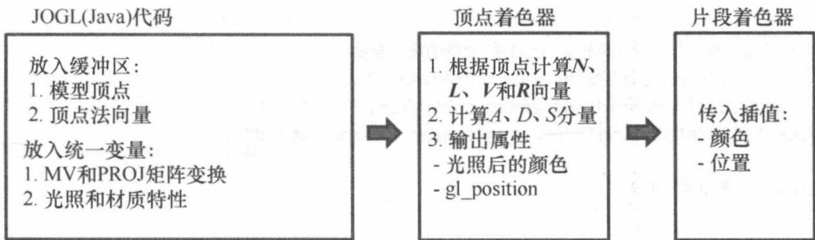


图 7.8 实现 Gouraud 着色

程序 7.1 位置光和 Gouraud 着色器下的环面

C++/OpenGL 应用程序

```
. . .
#include "Torus.h"
#include "Utils.h"
. . .
// 用于创建着色器和渲染程序的声明，如前
// VAO、两个 VBO 以及环面的声明，如前
// 环面与相机位置的声明和赋值，如前
// Utils.cpp 中现在已经添加有金、银、青铜材质
. . .
// 为 display() 函数分配变量
GLuint mvLoc, projLoc, nLoc;
```

```
// 着色器统一变量中的位置
GLuint globalAmbLoc, ambLoc, diffLoc, specLoc, posLoc, mAmbLoc, mDiffLoc, mSpecLoc, mShiLoc;

glm::mat4 pMat, vMat, mMat, mvMat, invTrMat;
glm::vec3 currentLightPos, lightPosV;   // 在模型和视觉空间中的光照位置，Vector3f 类型
float lightPos[3];                      // 光照位置的浮点数组

// 初始化光照位置
glm::vec3 initialLightLoc = glm::vec3(5.0f, 2.0f, 2.0f);

// 白光特性
float globalAmbient[4] = { 0.7f, 0.7f, 0.7f, 1.0f };
float lightAmbient[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
float lightDiffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
float lightSpecular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };

// 黄金材质特性
float* matAmb = Utils::goldAmbient();
float* matDif = Utils::goldDiffuse();
float* matSpe = Utils::goldSpecular();
float matShi = Utils::goldShininess();

void setupVertices(void) {
   // 该函数与之前章节中的相同，没有改动
   // 下面的部分在这里出现是为了更清晰，现在我们将真的使用法向量

   . . .
   glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
   glBufferData(GL_ARRAY_BUFFER, nvalues.size() * 4, &nvalues[0], GL_STATIC_DRAW);
}

void display(GLFWwindow* window, double currentTime) {
   // 清除深度缓冲区，如之前例子中一样载入渲染程序

   . . .
   // 用于模型-视图变换、投影以及逆转置(法向量)矩阵的统一变量
   mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
   projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
   nLoc = glGetUniformLocation(renderingProgram, "norm_matrix");

   // 初始化投影及视图矩阵，如前例

   . . .
   // 基于环面位置，构建模型矩阵
   mMat = glm::translate(glm::mat4(1.0f), glm::vec3(torLocX, torLocY, torLocZ));
   // 旋转环面以便更容易看到
   mMat *= glm::rotate(mMat, toRadians(35.0f), glm::vec3(1.0f, 0.0f, 0.0f));

   // 基于当前光源位置，初始化光照
   currentLightPos = glm::vec3(initialLightLoc.x, initialLightLoc.y, initialLightLoc.z);
   installLights(vMat);
   // 通过合并矩阵 v 和 m，创建模型-视图(MV)矩阵，如前
   mvMat = vMat * mMat;

   // 构建 MV 矩阵的逆转置矩阵，以变换法向量
   invTrMat = glm::transpose(glm::inverse(mvMat));

   // 将 MV、PROJ 以及逆转置(法向量)矩阵传入相应的统一变量
   glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
```

```
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    glUniformMatrix4fv(nLoc, 1, GL_FALSE, glm::value_ptr(invTrMat));

    // 在顶点着色器中，将顶点缓冲区(VBO #0)绑定到顶点属性#0
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    glEnableVertexAttribArray(0);

    // 在顶点着色器中，将法向缓冲区(VBO #2)绑定到顶点属性#1
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
    glEnableVertexAttribArray(1);

    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
    glDrawElements(GL_TRIANGLES, myTorus.getNumIndices(), GL_UNSIGNED_INT, 0);
}

void installLights(glm::mat4 vMatrix) {
    // 将光源位置转换为视图空间坐标，并存入浮点数组
    lightPosV = glm::vec3(vMatrix * glm::vec4(currentLightPos, 1.0));
    lightPos[0] = lightPosV.x;
    lightPos[1] = lightPosV.y;
    lightPos[2] = lightPosV.z;

    // 在着色器中获取光源位置和材质属性
    globalAmbLoc = glGetUniformLocation(renderingProgram, "globalAmbient");
    ambLoc = glGetUniformLocation(renderingProgram, "light.ambient");
    diffLoc = glGetUniformLocation(renderingProgram, "light.diffuse");
    specLoc = glGetUniformLocation(renderingProgram, "light.specular");
    posLoc = glGetUniformLocation(renderingProgram, "light.position");
    mAmbLoc = glGetUniformLocation(renderingProgram, "material.ambient");
    mDiffLoc = glGetUniformLocation(renderingProgram, "material.diffuse");
    mSpecLoc = glGetUniformLocation(renderingProgram, "material.specular");
    mShiLoc = glGetUniformLocation(renderingProgram, "material.shininess");

    // 在着色器中为光源与材质统一变量赋值
    glProgramUniform4fv(renderingProgram, globalAmbLoc, 1, globalAmbient);
    glProgramUniform4fv(renderingProgram, ambLoc, 1, lightAmbient);
    glProgramUniform4fv(renderingProgram, diffLoc, 1, lightDiffuse);
    glProgramUniform4fv(renderingProgram, specLoc, 1, lightSpecular);
    glProgramUniform3fv(renderingProgram, posLoc, 1, lightPos);
    glProgramUniform4fv(renderingProgram, mAmbLoc, 1, matAmb);
    glProgramUniform4fv(renderingProgram, mDiffLoc, 1, matDif);
    glProgramUniform4fv(renderingProgram, mSpecLoc, 1, matSpe);
    glProgramUniform1f(renderingProgram, mShiLoc, matShi);
}
// init() 以及 main() 函数如前
```

程序 7.1 中的很多元素我们都已经熟悉了。首先，定义了环面、光照和材质特性。接着将环面顶点以及相关法向量读入缓冲区。display()函数与之前程序中的类似，在这里不同的是它同时也将光照和材质信息传入顶点着色器。为了传入这些信息，它调用 installLights()，将光源在视觉空间中的位置，以及材质的 ADS 特性，读入相应的统一变量以供着色器使用。注意，我们提前定义了这些统一位置变量，以求更好的性能。

其中一个重要的细节是变换矩阵MV，用来将顶点位置移动到视觉空间，但它并不总能正确地将法向量也调整进视觉空间。直接对法向量应用MV矩阵不能保证法向量依然与物体表面垂直。正确的变换是MV的逆转置矩阵，在第3章“补充说明”中有描述。在程序7.1中，这个新增的矩阵叫作“invTrMat”，通过统一变量传入着色器。

变量lightPosV包含光源在相机空间中的位置。我们每帧只需要计算一次，因此我们在installLights()中[在display()中调用]而非着色器中计算。着色器在下方的续程序7.1中。其中顶点着色器使用了一些我们目前没有见过的符号。注意，在顶点着色器最后进行了向量加法——在第3章中有讲，并且在GLSL中可用。我们将会在展示着色器之后讨论其他符号。

续程序 7.1

顶点着色器

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec4 varyingColor;

struct PositionalLight
{ vec4 ambient;
  vec4 diffuse;
  vec4 specular;
  vec3 position;
};
struct Material
{ vec4 ambient;
  vec4 diffuse;
  vec4 specular;
  float shininess;
};
uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;    // 用来变换法向量

void main(void)
{ vec4 color;

  // 将顶点位置转换到视觉空间
  // 将法向量转换到视觉空间
  // 计算视觉空间光照向量(从顶点到光源)
  vec4 P = mv_matrix * vec4(vertPos,1.0);
  vec3 N = normalize((norm_matrix * vec4(vertNormal,1.0)).xyz);
  vec3 L = normalize(light.position - P.xyz);

  // 视觉向量等于视觉空间中的负顶点位置
  vec3 V = normalize(-P.xyz);

  // R是-L的相对于表面向量N的镜像
  vec3 R = reflect(-L,N);

  // 环境光、漫反射和镜面反射分量
  vec3 ambient = ((globalAmbient * material.ambient) + (light.ambient * material.ambient)).xyz;
  vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(dot(N,L), 0.0);
```

```
  vec3 specular =
     material.specular.xyz * light.specular.xyz * pow(max(dot(R,V), 0.0f), material.shininess);

  // 将颜色输出发送到片段着色器
  varyingColor = vec4((ambient + diffuse + specular), 1.0);

  // 将位置发送到片段着色器，如前
  gl_Position = proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

片段着色器

```
#version 430
in vec4 varyingColor;
out vec4 fragColor;

// 与顶点着色器相同的统一变量
// 但并不直接在当前片段着色器使用

struct PositionalLight
{ vec4 ambient;
  vec4 diffuse;
  vec4 specular;
  vec3 position;
};
struct Material
{ vec4 ambient;
  vec4 diffuse;
  vec4 specular;
  float shininess;
};
uniform vec4 globalAmbient;
uniform PositionalLight light;
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;

void main(void)
{ fragColor = varyingColor;
}
```

程序 7.1 的输出如图 7.9 所示。

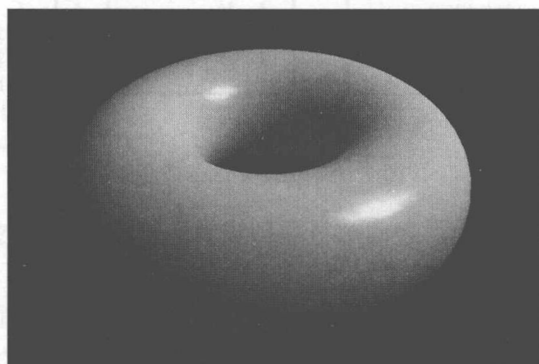


图 7.9 Gouraud 着色的环面

顶点着色器代码中有我们第一次使用了结构体语法的示例。GLSL“结构体”就像一个

数据类型，它有名称和一组字段。当使用结构体名称声明变量时，这个变量将包含结构体中声明的字段，并可以通过“.”语法访问字段。例如，变量“light”声明为“PositionalLight”类型，因此我们可以在其后引用其字段 light.ambient，light.diffuse 等。

还要注意字段选择器符号“.xyz”，我们在顶点着色器中的多个地方都使用了这种语法。这是将 vec4 转换为仅包含其前 3 个元素的等效 vec3 的快捷方式。

绝大多数光照计算发生在顶点着色器中。对于每个顶点，将适当的矩阵变换应用于顶点位置和相关的法向量，并计算用于光方向（$\boldsymbol{L}$）和反射（$\boldsymbol{R}$）的向量。然后执行 7.4 节中描述的 ADS 计算，得到每个顶点的颜色（代码中名为 varyingColor）。颜色作为正常光栅化过程的一部分进行插值。之后片段着色器仅作为简单传递。冗长的统一变量声明列表也在片段着色器中（由于前面第 4 章中描述的原因），但实际上并没有在那里使用它们。

注意 GLSL 函数 normalize()，它用来将向量转换为单位长度。正确地进行点积运算必须要先使用该函数。reflect()函数则计算一个向量基于另一个向量的反射。

图 7.9 输出的环面中有很明显的伪影。其镜面高光有着块状、面片感。这种伪影在物体移动时会更加明显（但我们在书中没法展示移动的物体）。

Gouraud 着色也容易受到其他伪影影响。如果镜面高光整个范围都在模型中的一个三角形内——即高光范围内一个模型顶点也没有——那么它可能不会被渲染出来。由于镜面反射分量是依顶点计算的，因此，当模型所有顶点都没有镜面反射分量时，其光栅化后的像素也不会有镜面反射光。

7.5.2 Phong 着色

Bui Tuong Phong 在犹他大学的研究生期间开发了一种平滑的着色算法，在 1973 年的论文[PH73]中对其进行了描述，并在[PH75]中发表。该算法的结构类似于 Gouraud 着色的算法，其不同之处在于光照计算是按像素而非顶点完成。由于光照计算需要法向量 $\boldsymbol{N}$ 和光向量 $\boldsymbol{L}$，但在模型中仅顶点包含这些信息，因此 Phong 着色通常使用巧妙的“技巧”来实现，其中 $\boldsymbol{N}$ 和 $\boldsymbol{L}$ 在顶点着色器中进行计算，并在光栅化期间插值。图 7.10 概述了此策略。

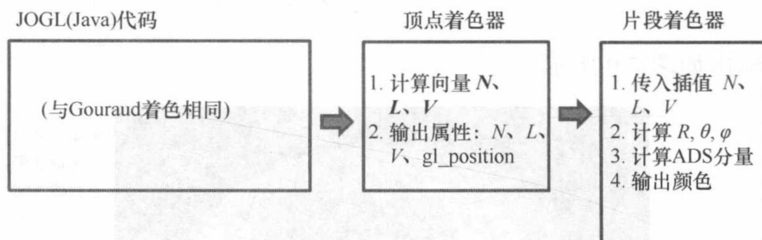


图 7.10 实现 Phong 着色

C++/OpenGL 代码完全如前。之前部分在顶点着色器中完成的过程现在回放入片段着色器中进行。法向量插值的效果如图 7.11 所示。

现在我们已经准备好使用 Phong 着色实现位置光照射下的环面了。大多数代码与实现 Gouraud 着色的代码相同。由于 C++/OpenGL 代码完全没有改变，在此我们只展示修改过的顶点着色器和片段着色器，见程序 7.2。程序 7.2 的输出如图 7.12 所示，Phong 着色修正了 Gouraud 着色中出现的伪影。

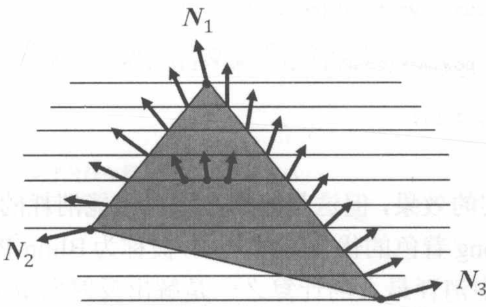
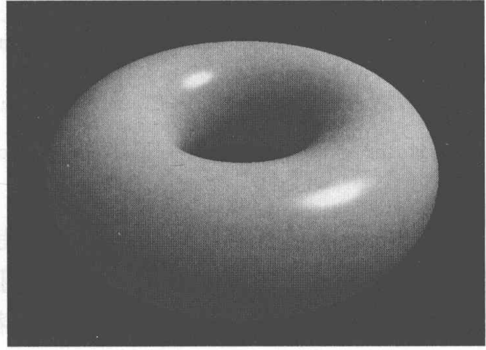


图 7.11 法向量插值


图 7.12 Phong 着色的环面

程序 7.2 Phong 着色的环面

顶点着色器

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;
out vec3 varyingNormal;       // 视觉空间顶点法向量
out vec3 varyingLightDir;     // 指向光源的向量
out vec3 varyingVertPos;      // 视觉空间中的顶点位置

// 结构体和统一变量与 Gouraud 着色相同
. . .
void main(void)
{ // 输出顶点位置、光照方向和法向量到光栅器以进行插值
  varyingVertPos=(mv_matrix * vec4(vertPos,1.0)).xyz;
  varyingLightDir = light.position - varyingVertPos;
  varyingNormal=(norm_matrix * vec4(vertNormal,1.0)).xyz;

  gl_Position=proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

片段着色器

```
#version 430
in vec3 varyingNormal;
in vec3 varyingLightDir;
in vec3 varyingVertPos;
out vec4 fragColor;

// 结构体和统一变量与 Gouraud 着色相同
. . .
void main(void)
{ // 正规化光照向量、法向量、视觉向量
  vec3 L = normalize(varyingLightDir);
  vec3 N = normalize(varyingNormal);
  vec3 V = normalize(-varyingVertPos);

  // 计算光照向量基于 N 的反射向量
  vec3 R = normalize(reflect(-L, N));
  // 计算光照与平面法向量间的角度
  float cosTheta = dot(L,N);
  // 计算视觉向量与反射光向量的角度
  float cosPhi = dot(V,R);

  // 计算 ADS 分量(按像素)，并合并以构建输出颜色
```

```
    vec3 ambient = ((globalAmbient * material.ambient) + (light.ambient * material.ambient)).xyz;
    vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(cosTheta,0.0);
    vec3 specular =
        light.specular.xyz * material.specular.xyz * pow(max(cosPhi,0.0), material.shininess);

    fragColor = vec4((ambient + diffuse + specular), 1.0);
}
```

虽然 Phong 着色有着比 Gouraud 着色更真实的效果，但这是建立在增大性能消耗的基础上的。James Blinn 在 1977 年提出了一种对于 Phong 着色的优化方法[BL77]，被称为 Blinn-Phong 反射模型。这种优化是基于观察到 Phong 着色中消耗最大的计算之一是解出反射向量 $\boldsymbol{R}$。

Blinn 发现向量 $\boldsymbol{R}$ 在计算过程中并不是必需的——$\boldsymbol{R}$ 只是用来计算角 φ 的手段。角 φ 的计算可以不用向量 $\boldsymbol{R}$，而通过 $\boldsymbol{L}$ 与 $\boldsymbol{V}$ 的角平分线向量 $\boldsymbol{H}$ 得到。如图 7.13 所示，$\boldsymbol{H}$ 和 $\boldsymbol{N}$ 之间的角 α 刚好等于 $1/2(\varphi)$。虽然 α 与 φ 不同，但 Blinn 展示了使用 α 代替 φ 就已经可以获得足够好的结果。

角平分线向量可以简单地使用 $\boldsymbol{L}+\boldsymbol{V}$ 得到（见图 7.14），之后 $\cos(\alpha)$ 可以通过 $\hat{\boldsymbol{H}} \bullet \hat{\boldsymbol{N}}$ 的点积计算。

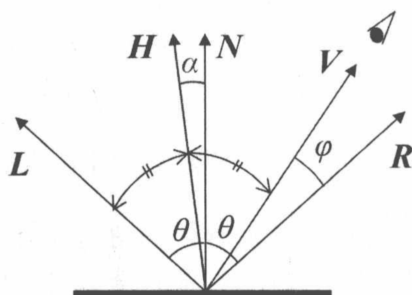


图 7.13 Blinn-Phong 反射

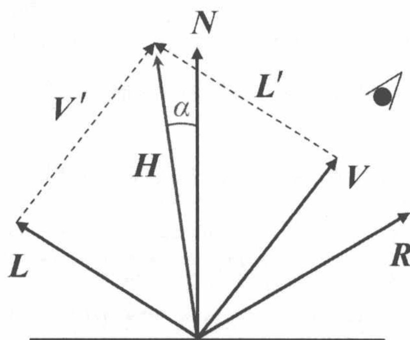


图 7.14 Blinn-Phong 计算

这些计算可以在片段着色器中进行，甚至为了性能考虑（经过一些调整）也可以在顶点着色器中进行。图 7.15 展示了使用 Blinn-Phong 着色的环面。它在图形质量上几乎与 Phong 渲染相同，同时节省了大量性能损耗。

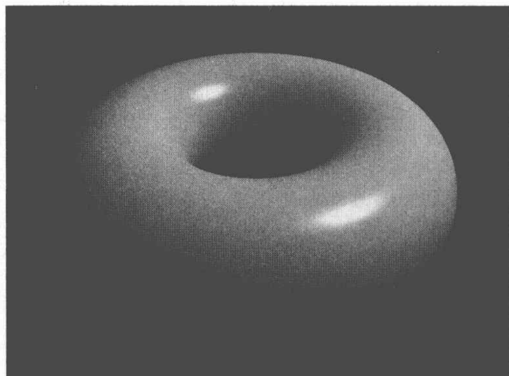


图 7.15 Blinn-Phong 着色的环面

程序 7.3 中展示了修改后顶点着色器和片段着色器，它们用来将程序 7.2 中的 Phong 着

色示例转换为 Blinn-Phong 着色。C++ / OpenGL 代码与之前一样没有变化。

程序 7.3 Blinn-Phong 着色的环面

顶点着色器

```
. . .
// 角平分线向量 H 作为新增的输出
out vec3 varyingHalfVector;

. . .
void main(void)
{ // 与之前的计算相同，增加了 L+V 的计算
  varyingHalfVector = (varyingLightDir + (-varyingVertPos)).xyz;

  // (其余顶点着色器代码没有改动)
}
```

片段着色器

```
. . .
in vec3 varyingHalfVector;
. . .
void main(void)
{ // 注意，现在已经不需要在片段着色器中计算 R
  vec3 L = normalize(varyingLightDir);
  vec3 N = normalize(varyingNormal);
  vec3 V = normalize(-varyingVertPos);
  vec3 H = normalize(varyingHalfVector);

  . . .
  // 计算法向量 N 与角平分线向量 H 之间的角度
  float cosPhi = dot(H,N);

  // 角平分线向量 H 已经在顶点着色器中计算过，并在光栅器中进行过插值
  vec3 ambient = ((globalAmbient * material.ambient) + (light.ambient * material.ambient)).xyz;
  vec3 diffuse = light.diffuse.xyz * material.diffuse.xyz * max(cosTheta,0.0);
  vec3 specular =
     light.specular.xyz * material.specular.xyz * pow(max(cosPhi,0.0), material.shininess*3.0);
     // 最后乘以 3.0 作为改善镜面高光的微调
  fragColor = vec4((ambient + diffuse + specular), 1.0);
}
```

图 7.16（见彩插）所示的两个例子展示了 Phong 着色应用在比较复杂的外部软件生成模

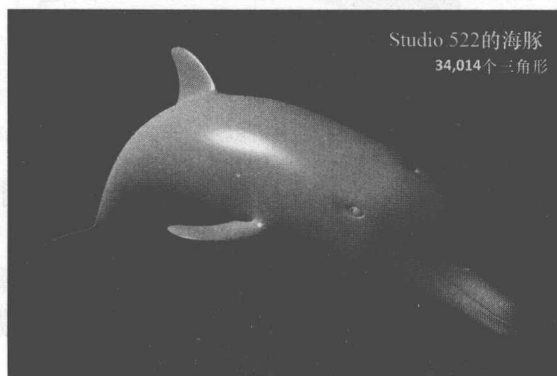


图 7.16　Phong 着色的外部模型

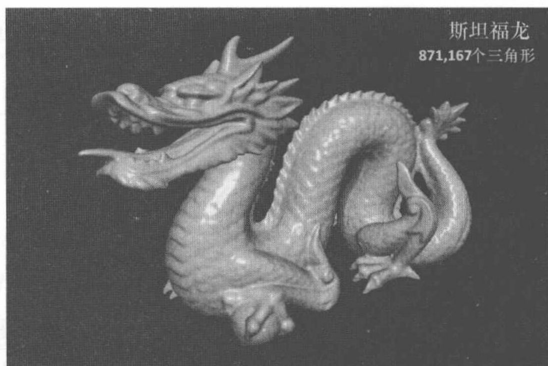


图 7.16 Phong 着色的外部模型（续）

型上所产生的效果。图 7.16 上图展示了 Jay Turberville 在 Studio 522 Productions [TU16]创建的 OBJ 格式海豚模型的渲染图。图 7.16 下图是著名的“斯坦福龙”的渲染，斯坦福龙是 1996 年对一个小模型进行 3D 扫描所得到的模型[ST96]。两个模型都使用我们放在“Utils.cpp”文件中的“黄金”材质进行渲染。斯坦福龙因其大小而被广泛用于测试图形算法和硬件——它包含超过 800 000 个三角形。

7.6 结合光照与纹理

目前为止，在光照模型中，都是假设我们使用按 ADS 定义的光源，照亮按 ADS 定义材质的物体。但是，正如我们在第 5 章中所讲的，某些对象的表面可能会指定纹理图像。因此，我们需要一种方法来结合采样纹理所得的颜色和光照模型产生的颜色。

我们结合光照和纹理的方式取决于物体的特性以及其纹理的目的。这里有多种情况，其中常见的有：

- 纹理图像很写实地反映了物体真实的表面外观；
- 物体同时具有材质和纹理；
- 材质包括了阴影和反射信息（在第 8 章、第 9 章中）；
- 有多种光和/或多个纹理。

我们先来观察第一种情景，物体拥有一个简单的纹理，同时我们对它进行光照。实现这种光照的一种简单方法是在片段着色器中完全将材质特性去除掉，之后使用纹理取样所得纹理颜色代替材质的 ADS 值。下面的伪代码展示了这种策略：

```
fragColor = textureColor * ( ambientLight + diffuseLight ) + specularLight
```

这种策略下，纹理颜色影响了环境光和漫反射分量，而镜面反射颜色仅由光源决定。镜面反射分量仅由光源决定是一种很常见的做法，尤其是对于金属或“闪亮”的表面。但是，对于不那么闪亮的表面，如织物或未上漆的木材（甚至一小部分金属，如黄金），其镜面高光部分都应当包含物体表明颜色。在这些情况下，之前的策略应该做适当微调：

```
fragColor = textureColor * ( ambientLight + diffuseLight + specularLight )
```

同时也有一些情况下，物体本身具有 ADS 材质，并伴有纹理图像。如银质物体使用纹理为表面添加一些氧化痕迹。在这些情况下，如之前章节中所讲过的，既用到光照又用到材质的标准 ADS 模型就可以与纹理颜色相结合，并加权求和。如：

```
textureColor = texture(sampler, texCoord)
lightColor = (ambLight * ambMaterial) + (diffLight * diffMaterial) + specLight
fragColor = 0.5 * textureColor + 0.5 * lightColor
```

这种策略结合了光照、材质、纹理，并能够扩展到多个光源以及多种材质的情况。如：

```
texture1Color = texture(sampler1, texCoord)
texture2Color = texture(sampler2, texCoord)

light1Color = (ambLight1 * ambMaterial) + (diffLight1 * diffMaterial) + specLight1
light2Color = (ambLight2 * ambMaterial) + (diffLight2 * diffMaterial) + specLight2

fragColor = 0.25 * texture1Color
          + 0.25 * texture2Color
          + 0.25 * light1Color
          + 0.25 * light2Color
```

图 7.17（见彩插）展示了拥有 UV 映射纹理图像（来自 Jay Turberville[TU16]）的 Studio 522 海豚，以及我们之前在第 6 章见过的 NASA 航天飞机模型。这两个有纹理的模型都使用了增强后的 Blinn-Phong 光照，没有使用材质，并在镜面高光中仅使用光照进行计算。在这两幅图中，片段着色器中颜色相关的计算为：

```
vec4 texColor = texture(sampler, texCoord);
fragColor = texColor * (globalAmbient + lightAmb + lightDiff * max(dot(L,N),0.0))
            + lightSpec * pow(max(dot(H,N),0.0), matShininess*3.0);
```

注意，计算过程中 fragColor 可能产生大于 1.0 的值。在这种情况下，OpenGL 会将它限制回 1.0。

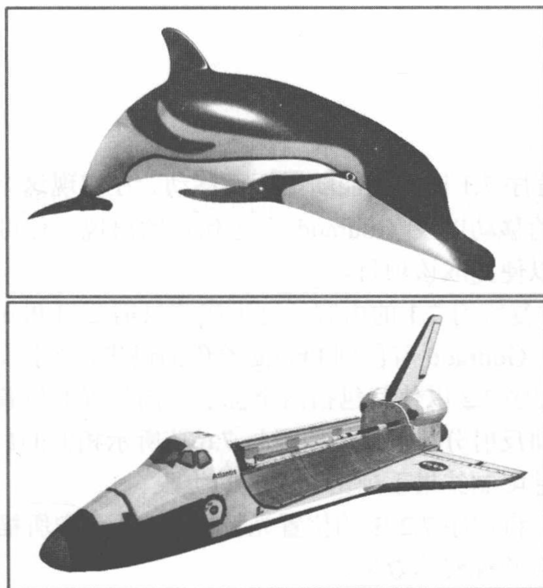


图 7.17 结合光照与纹理

补充说明

图 7.7 所展示的面片着色的环面是通过在顶点着色器和片段着色器中，将“flat”插值限定符添加到相应的法向量属性声明中得到的。这样会使得光栅器不对所限定的变量进行插值，而是直接将相同的值赋给每个片段（在默认情况下，它会选择三角形第一个顶点上的值）。在 Phong 着色示例代码中，可以通过如下修改实现面片着色：

```
在顶点着色器中
flat out vec3 varyingNormal;
在片段着色器中
flat in vec3 varyingNormal;
```

我们还没有讨论的一类很重要的光是分布式光(distributed light)(或区域光(area light))，这种光的光源是一片区域而非一个单点。它在现实世界相对应的例子是通常在办公室或教室中的日光灯管。有兴趣的读者可以在[MH02]找到更多有关区域光的详细信息。

历史记录

在本章中我们过度简化了 Gouraud 和 Phong 的一些术语。Gouraud 着色归功于 Gouraud——通过计算顶点上光的强度并使用光栅器对光强进行插值以生成平滑的曲面外观（有时也被称为“平滑着色”）。Phong 着色则归功于 Phong，这是另一种平滑着色，对法向量插值并计算每个像素的光照。Phong 同时也被认为是成功将镜面高光纳入平滑着色的先驱者。因此，ADS 光照模型在计算机图形学中也通常被称为 Phong 反射模型。因此，我们例子中的 Gouraud 着色准确地来说是使用了 Phong 反射模型的 Gouraud 着色。由于 Phong 的反射模型在 3D 图形编程中非常普及，通常 Gouraud 着色模型都是在 Phong 反射模型中进行展示。不过这可能会引起误会，因为原本 Gouraud 在 1971 年的工作中并没有任何镜面反射分量。

习题

7.1 （项目）修改程序 7.1 以使光能随鼠标而移动。在实现这个功能之后，四处移动鼠标，并记录下镜面高光的移动以及 Gouraud 着色伪影的出现。你可能会需要在光源处渲染一个点（或者小物体）以便完成该项目。

7.2 在程序 7.2 中重复练习 7.1 的内容。这里应该只需要将 Phong 着色的着色器放入练习 7.1 的解决方案中。从 Gouraud 着色到 Phong 着色的进步在光四处移动时应当更明显。

7.3 （项目）修改程序 7.2 以使其包括两个位于不同位置的位置光。片段着色器需要混合每个光的漫反射和镜面反射分量。尝试使用与 7.6 节所示相似的加权求和方法。你可以尝试简单地将它们加起来并限制结果不超出光照值的上限。

7.4 （研究和项目）将程序 7.2 中的位置光替换为 7.2 节中所描述的探照灯。尝试设置不同的遮光角、衰减指数并观察其效果。

参考资料

[BA16] N. Barradeu, accessed October 2018.

[BL77] J. Blinn, "Models of Light Reflection for Computer Synthesized Pictures," *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, 1977.

[DI16] *Luxo Jr.* (Pixar – copyright held by Disney), accessed October 2018.

[GO71] H. Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers* C-20, no. 6 (June 1971).

[MH02] T. Akenine-Mö ller and E. Haines, *Real-Time Rendering*, 2nd ed. (A. K. Peters, 2002).

[PH73] B. Phong, "Illumination of Computer-Generated Images" (PhD thesis, University of Utah, 1973).

[PH75] B. Phong, "Illumination for Computer Generated Pictures," *Communications of the ACM* 18, no. 6 (June 1975): 311-317.

[ST96] Stanford Computer Graphics Laboratory, 1996, accessed October 2018.

[TU16] J. Turberville, Studio 522 Productions, Scottsdale, AZ.

第 8 章　阴影

8.1　阴影的重要性

在第 7 章中，我们学会了如何为 3D 场景添加光照。但是，我们并没有真的添加光线，而是模拟光照在物体上的效果——使用 ADS 模型——并相应地调整这些物体的绘制方式。

当我们用这种方法照亮同一个场景中的多个物体时，它的局限性就体现出来了。考虑图 8.1 所示的场景，其中包含了砖块纹理环面以及地平面（地平面是一个巨大立方体的顶部，使用了来自[LU16]的草地纹理）。

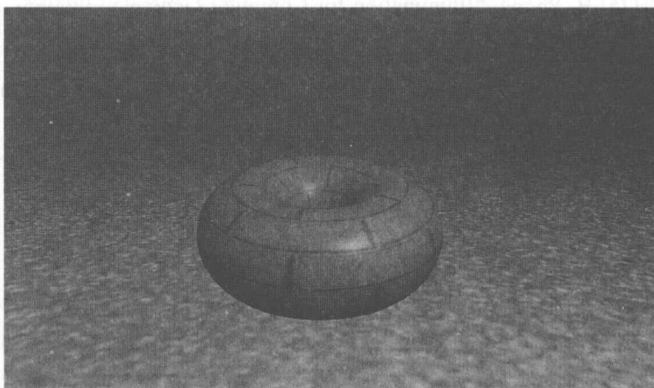


图 8.1　没有阴影的场景

一眼望去我们的场景好像没问题。但是，仔细观察会发现有什么重要的东西没有出现。具体来说，就是我们没有办法分辨出环面距离它下方纹理立方体的距离。环面究竟是浮在立方体上面呢，还是放置在立方体顶部呢？

我们无法回答这个问题的原因正是因为场景中缺乏阴影。我们期望看到阴影，因为大脑需要通过阴影，才能针对我们所看到的物体以及他们的位置关系构建完整的心理模型。

考虑图 8.2 所示的同样的场景，不过添加了阴影。现在就很明显了，左图中环面放在地平面上；而右图中，环面则浮于其上。

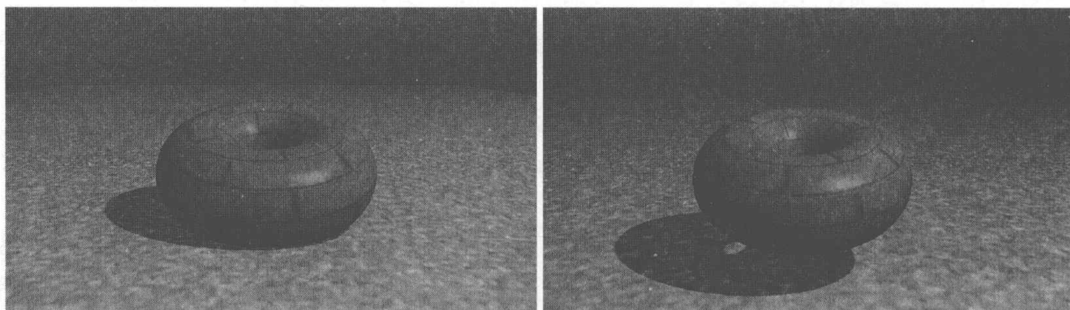


图 8.2　带阴影的光照

8.2 投影阴影

为了给 3D 场景添加阴影，人们设计了许多有趣的方法。其中一种很适合在地平面上（如图 8.1 所示）绘制阴影，又相对不需要太大计算代价的方法，叫作投影阴影（projective shadows）。给定一个位于（X_L,Y_L,Z_L）的点光源、一个需要渲染的物体以及一个投射阴影的平面，可以通过生成一个变换矩阵，将物体上的点（X_W,Y_W,Z_W）变换为相应阴影在平面上的点（$X_S,0,Z_S$）。之后将其生成的“阴影多边形”绘制出来，通常使用暗色物体与地平面纹理混合作为其纹理，如图 8.3 所示。

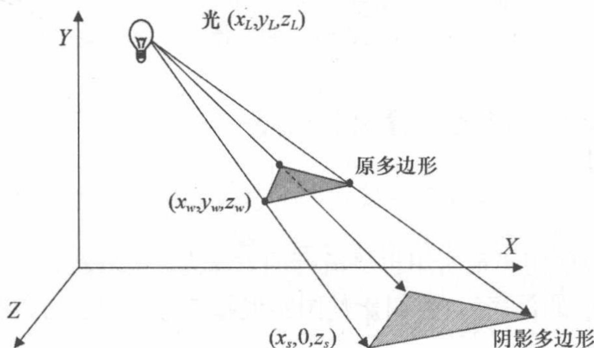


图 8.3 投影阴影

使用投影阴影进行投射的优点是它的高效和易于实现。但是，它仅适用于平坦表面——这种方法无法投射阴影于曲面或其他物体。即使如此，它仍然适用于有室外场景并对性能要求较高的应用，很多游戏中的场景都属于这类。

投影阴影变换矩阵的发展在[BL88], [AS14]以及[KS16]中有讨论。

8.3 阴影体

Franklin C. Crow 在 1977 年提出了另一个重要的方法，这个方法先找到被物体阴影覆盖的阴影体，之后减少视体与阴影体相交部分中的多边形的颜色强度。图 8.4 展示了阴影体中的立方体，因此，立方体绘制时会更暗。

阴影体的优点在于其高度准确，比起其他方法来更不容易产生伪影。但是，计算出阴影体以及每个多边形是否在其中这件事，即使对于现代 GPU 来说，计算代价也很大。几何着色器可以用于计算阴影体，模板缓冲区①可以用于判断像素是否在阴影体内。有些显卡对于特定的阴影体操作优化提供了硬件支持。

① 模板缓冲区是通过 OpenGL 访问的第三个缓冲区——在颜色缓冲区和 Z 缓冲区之后。本书中不对模板缓冲区进行讲解。——译者注

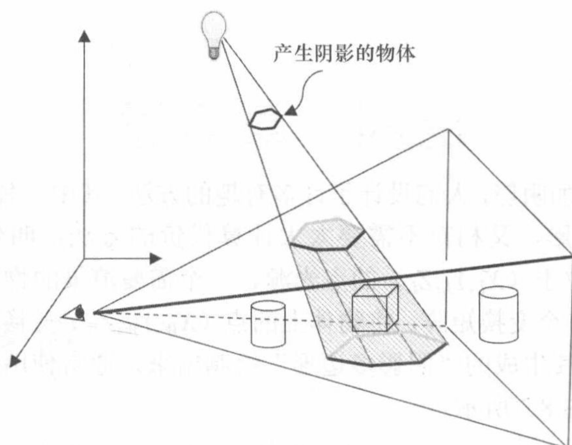


图 8.4 阴影体

8.4 阴影贴图

阴影贴图是用于投射阴影最实用也最流行的方法之一。虽然它并不总是像阴影体一样准确（且通常伴随着讨厌的伪影），但阴影贴图实现起来更简单，可以在各种情况下使用，并享有强大的硬件支持。

如果我们不在这里澄清前一段中的“更简单”这个词，那将是我们的疏忽。虽然阴影贴图比阴影体（在概念和实践中）更简单，但它绝不“简单”！对学生来说，通常在 3D 图形课程中最难实现的技术之一就是阴影贴图。着色器程序本质上很难调试，阴影贴图需要几个组件和着色器模块的完美协调。请注意，通过使用前面 2.2 节中描述的调试工具，可以极大地促进阴影贴图的成功实现。

阴影贴图基于一个非常简明的想法：光线无法看到的任何东西都在阴影中。也就是说，如果对象＃1 阻挡光到达对象＃2，等同于光不能“看到”对象＃2。

这个想法的强大之处在于我们已经有了方法来确定物体是否可以被“看到”——使用 Z 缓冲区的隐藏面消除算法（HSR），如 2.1.7 节所述。因此，计算阴影的策略是，暂时将摄像机移动到光的位置，应用 Z 缓冲区 HSR 算法，然后使用生成的深度信息来计算阴影。

因此，渲染场景需要两轮：第 1 轮从灯光的角度渲染场景（但实际上没有将其绘制到屏幕上），第 2 轮从摄像机的角度渲染场景。第 1 轮的目的是从光的角度生成 Z 缓冲区。完成第 1 轮之后，我们需要保留 Z 缓冲区并使用它来帮助我们在第 2 轮生成阴影。第 2 轮实际绘制场景。

我们的策略可以更加精炼。

- （第 1 轮）从灯光的位置渲染场景。然后，对于每个像素，深度缓冲区包含光与最近的对象之间的距离。
- 将深度缓冲区复制到单独的“阴影缓冲区”。
- （第 2 轮）正常渲染场景。对于每个像素，在阴影缓冲区中查找相应的位置。如果相机到渲染点的距离大于从阴影缓冲区检索到的值，则在该像素处绘制的对象离光

线的距离，比离光线最近的对象更远，因此该像素处于阴影中。

当发现像素处于阴影中时，我们需要使其更暗。一种简单而有效的方法是仅渲染其环境光，忽略其漫反射和镜面反射分量。

上述方法通常被称为“阴影缓冲区”。而当我们在第二步中，将深度缓冲区复制到纹理中，则称为“阴影贴图”。当纹理对象用于储存阴影深度信息时，我们称其为*阴影纹理*，OpenGL 通过 sampler2DShadow 类型支持阴影纹理（稍后讨论）。这样，我们就可以利用片段着色器中纹理单元和采样器变量（即“纹理贴图”）的硬件支持功能，在第 2 轮快速执行深度查找。我们现在修改的策略是：

- （第 1 轮）与之前相同；
- 将深度缓冲区的内容复制进纹理对象；
- （第 2 轮）与之前相同，不过阴影缓冲区变为阴影纹理。

现在我们来实现这些步骤。

8.4.1 阴影贴图（第 1 轮）——从光源位置“绘制”物体

在第一步中，我们首先将相机移动到灯光的位置然后渲染场景。我们的目标不是在显示器上实际绘制场景，而是完成足够的渲染过程以正确填充深度缓冲区。因此，没有必要为像素生成颜色，我们的第一遍将仅使用顶点着色器，但片段着色器不执行任何操作。

当然，移动相机需要构建适当的观察矩阵。根据场景的内容，我们需要在光源处依合适的方向来看场景。通常，我们希望此方向朝向最终在第 2 轮中呈现的区域。

这个方向通常依场景而定——在我们的场景中，我们通常会将相机从光源指向原点。

第 1 轮中有几个需要处理的重要细节。

- 配置缓冲区和阴影纹理。
- 禁用颜色输出。
- 从光源到视野中的物体构建一个 LookAt 矩阵。
- 启用 GLSL 第 1 轮着色器程序，该程序仅包含图 8.5 中的简单顶点着色器，准备接收 MVP 矩阵。在这种情况下，MVP 矩阵将包括对象的模型矩阵 $\boldsymbol{M}$、前一步中计算的 LookAt 矩阵（作为观察矩阵 $\boldsymbol{V}$），以及透视矩阵 $\boldsymbol{P}$。我们将该 MVP 矩阵称为“shadowMVP”，因为它是基于光而不是相机的观察点。由于实际上没有显示来自光源的视图，因此第 1 轮着色器程序的片段着色器不会执行任何操作。

```
#version 430            //顶点着色器
layout (location=0) in vec3 vertPos;
uniform mat4 shadowMVP;

void main(void)
{    gl_Position = shadowMVP * vec4(vertPos,1.0);
}
```

```
#version 430            //片段着色器
void main(void)  { }
```

图 8.5 阴影贴图第 1 轮的顶点着色器和片段着色器

- 为每个对象创建 shadowMVP 矩阵，并调用 glDrawArrays()。第 1 轮中不需要包含纹理或光照，因为对象不会渲染到屏幕上。

8.4.2 阴影贴图（中间步骤）——将 Z 缓冲区复制到纹理

OpenGL 提供了两种将 Z 缓冲区深度数据放入纹理单元的方法。第一种方法是生成空阴影纹理，然后使用命令 glCopyTexImage2D()将活动深度缓冲区复制到阴影纹理中。

第二种方法是在第 1 轮中构建一个“自定义帧缓冲区”（而不是使用默认的 Z 缓冲区），并使用命令 glFrameBufferTexture()将阴影纹理附加到它上面。OpenGL 在 3.0 版中引入该命令，以进一步支持阴影纹理。使用这种方法时，无须将 Z 缓冲区“复制”到纹理中，因为缓冲区已经附加了纹理，深度信息由 OpenGL 自动放入纹理中。我们将在实现中使用这种方法。

8.4.3 阴影贴图（第 2 轮）——渲染带阴影的场景

第 2 轮中的大部分内容与我们在第 7 章中看到的类似，即我们在这里渲染完整的场景及其中的所有物体，以及光照、材质和装饰场景中物体的纹理。同时，我们还需要添加必要的代码，以确定每个像素是否在阴影中。

第 2 轮的一个重要特征是它使用了两个 MVP 矩阵。一个是将对象坐标转换为屏幕坐标的标准 MVP 矩阵（如我们之前的大多数示例所示）。另一个是在第 1 轮中生成的 shadowMVP 矩阵，用于从光源的角度进行渲染——现在将在第 2 轮中用于从阴影纹理中查找深度信息。

在第 2 轮中，从纹理贴图尝试查找像素时，情况比较复杂。OpenGL 相机使用[−1…+ 1]坐标空间，而纹理贴图使用[0…1]空间。常见的解决方案是构建一个额外的矩阵变换，通常称为 ***B***，它将用于从摄像机空间到纹理空间的转换（或“偏离”，biases，因此名称）。得到 ***B*** 的过程很简单——先缩放为 1/2，再平移 1/2。

矩阵 ***B*** 如下：

$$\boldsymbol{B}=\begin{bmatrix}0.5 & 0 & 0 & 0.5\\ 0 & 0.5 & 0 & 0.5\\ 0 & 0 & 0.5 & 0.5\\ 0 & 0 & 0 & 1\end{bmatrix}$$

之后将 ***B*** 合并入 shadowMVP 矩阵以备在第 2 轮中使用，如下：

$$\text{shadowMVP2}=[\boldsymbol{B}][\text{shadowMVP}_{(pass1)}]$$

假设我们使用阴影纹理附加到我们的自定义帧缓冲区的方法，OpenGL 提供了一些相对简单的工具，用于确定绘制对象时，像素是否处于阴影中。以下是第二阶段处理的详细信息摘要。

- 构建变换矩阵 B，用于从光照空间转换到纹理空间[更合适在 init()中进行]。
- 启用阴影纹理以进行查找。
- 启用颜色输出。
- 启用 GLSL 第 2 轮渲染程序，包含顶点着色器和片段着色器。

- 根据摄像机位置（正常）为正在绘制的对象构建 MVP 矩阵。
- 构建 shadowMVP2 矩阵（包含 ***B*** 矩阵，如前所述）——着色器将需要用它查找阴影纹理中的像素坐标。
- 将生成的矩阵变换发送到着色器统一变量。
- 像往常一样启用包含顶点、法向量和纹理坐标（如果使用）的缓冲区。
- 调用 glDrawArrays()。

除了渲染任务外，顶点和片段着色器还需要额外承担一些任务。

- 顶点着色器将顶点位置从相机空间转换为光照空间，并将结果坐标发送到顶点属性中的片段着色器，以便对它们进行插值。这样片段着色器可以从阴影纹理中检索正确的值。
- 片段着色器调用 textureProj()函数，该函数返回 0 或 1，指示像素是否处于阴影中（所涉及的机制将在后面解释）。如果它在阴影中，则着色器通过剔除其漫反射和镜面反射分量来输出更暗的像素。

阴影贴图是一种常见任务，因此 GLSL 为其提供了一种特殊类型的采样器变量，称为 sampler2DShadow（如前所述），可以附加到 C++ / OpenGL 应用程序中的阴影纹理。textureProj()函数用于从阴影纹理中查找值，它类似于我们之前在第 5 章中看到的 texture()，其区别是除了 textureProj()函数使用 vec3 来索引纹理而不是通常的 vec2。由于像素坐标是 vec4，因此需要将其投影到 2D 纹理空间上，以便在阴影纹理贴图中查找深度值。正如我们将在下面看到的，textureProj()为完成了这些功能。

顶点着色器和片段着色器代码的其余部分实现了 Blinn-Phong 着色。这些着色器如图 8.6 和图 8.7 所示，并增加了阴影贴图的代码。

```
#version 430
layout (location=0) in vec3 vertPos;
layout (location=1) in vec3 vertNormal;

out vec3 varyingNormal, varyingLightDir, varyingVertPos, varyingHalfVec;
out vec4 shadow_coord;

struct PositionalLight { vec4 ambient, diffuse, specular;  vec3 position; };
struct Material { vec4 ambient, diffuse, specular;  float shininess; };
uniform vec4 globalAmbient;
uniform PositionalLight light;      // 假设光源位置以视觉空间坐标表示
uniform Material material;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
uniform mat4 shadowMVP2;
layout (binding=0) uniform sampler2DShadow shTex;

void main(void)
{    varyingVertPos = (mv_matrix * vec4(vertPos,1.0)).xyz;
     varyingLightDir = light.position - varyingVertPos;
     varyingNormal = (norm_matrix * vec4(vertNormal,1.0)).xyz;
     varyingHalfVec = (varyingLightDir - varyingVertPos).xyz;
     shadow_coord = shadowMVP2 * vec4(vertPos,1.0);
     gl_Position = proj_matrix * mv_matrix * vec4(vertPos,1.0);
}
```

图 8.6　阴影贴图第 2 轮顶点着色器

让我们更仔细地研究一下如何使用 OpenGL 来执行正在渲染的像素和阴影纹理中的值之间的深度比较。首先，从顶点着色器开始，在模型空间中使用顶点坐标，我们将其与

shadowMVP2 相乘以生成阴影纹理坐标，这些坐标对应于投影到光照空间中的顶点坐标，是之前从光源的视角生成的。经过插值后的（3D）光照空间坐标（x, y, z）在片段着色器中使用如下。z 分量表示从光到像素的距离。（x, y）分量用于检索存储在（2D）阴影纹理中的深度信息。将该检索的值（到最靠近光的物体的距离）与 z 进行比较。该比较产生“二元”结果，告诉我们我们正在渲染的像素是否比最接近光的物体离光更远（即像素是否处于阴影中）。

假设光源位置以视觉空间坐标表示。

与顶点着色器相同的结构体和统一变量。

如果我们在 OpenGL 中使用前面介绍过的 glFrameBufferTexture()并启用深度测试，然后使用片段着色器（见图 8.7）的 sampler2DShadow 和 textureProj()，所渲染的结果将完全满足我们的需求。即 textureProj()将输出 0.0 或 1.0，具体取决于深度比较。基于此值，当像素离光源比离光源最近的物体更远时，我们可以在片段着色器中忽略漫反射和镜面反射分量，从而有效地创建阴影。概述如图 8.8 所示。

```
#version 430
in vec3 varyingNormal, varyingLightDir, varyingVertPos, varyingHalfVec;
in vec4 shadow_coord;
out vec4 fragColor;

// 与顶点着色器相同的结构体和统一变量
...
void main(void)
{   vec3 L = normalize(varyingLightDir);
    vec3 N = normalize(varyingNormal);
    vec3 V = normalize(-varyingVertPos);
    vec3 H = normalize(varyingHalfVec);

    float notInShadow = textureProj(shTex, shadow_coord);

    fragColor = globalAmbient * material.ambient + light.ambient * material.ambient;
    If (notInShadow == 1.0)
    {   fragColor += light.diffuse * material.diffuse * max(dot(L,N),0.0)
            + light.specular * material.specular
            * pow(max(dot(H,N),0.0),material.shininess*3.0);
    }
}
```

图 8.7 阴影贴图第 2 轮片段着色器

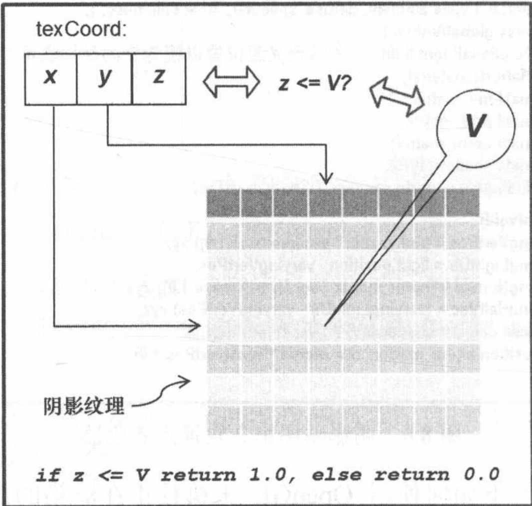


图 8.8 自动深度比较

我们现在准备构建 C++ / OpenGL 应用程序以使用上述着色器。

8.5 阴影贴图示例

考虑图 8.9 中包含环面和金字塔的场景。位置光源放置在左侧（注意镜面高光）。

金字塔应该在环面上投下阴影。

为了阐明示例的开发，我们的第一步是将第 1 轮渲染到屏幕以确保它正常工作。为此，我们将临时添加一个简单的片段着色器（它不会包含在最终版本中）并在第 1 轮中仅输出一种固定颜色（如红色）；例如：

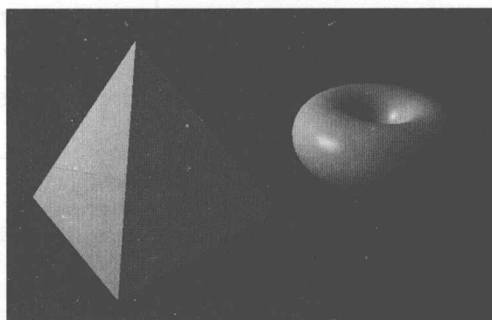


图 8.9 有光照无阴影的场景

```
#version 430
out vec4 fragColor;
void main(void)
{ fragColor = vec4(1.0, 0.0, 0.0, 0.0);
}
```

让我们假设场景的原点位于图的中心在金字塔和环面之间。在第 1 轮中，我们将相机放在光源的位置（图 8.10 中的左图）并指向（0,0,0）。然后我们用红色绘制对象，它会产生如图 8.10（见彩插）右图所示的输出。注意金字塔顶部附近的环面——这个制高点附近的环面部分位于金字塔后面。

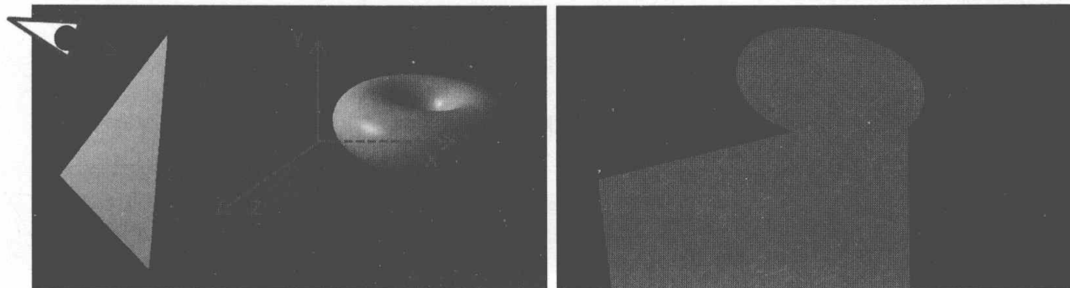


图 8.10 第 1 轮：场景（左）和从光源视角渲染的场景（右）

包含光照与阴影贴图的完整第 2 轮 C++/OpenGL 代码见程序 8.1。

程序 8.1 阴影贴图

```
// 大部分与之前相同。高亮部分代码是新加入的，用以实现阴影
// 实现光照所需的大部分引用需要在代码开始引入，与之前相同
// 因此不在这里重复

// 在这里定义渲染程序所用的变量、缓冲区、着色器源代码等
. . .
ImportedModel pyramid("pyr.obj");       // 定义金字塔
Torus myTorus(0.6f, 0.4f, 48);          // 定义环面
int numPyramidVertices, numTorusVertices, numTorusIndices;
```

```
. . .
// 环面、金字塔、相机和光源的位置
glm::vec3 torusLoc(1.6f, 0.0f, -0.3f);
glm::vec3 pyrLoc(-1.0f, 0.1f, 0.3f);
glm::vec3 cameraLoc(0.0f, 0.2f, 6.0f);
glm::vec3 lightLoc(-3.8f, 2.2f, 1.1f);

// 场景中所使用白光的属性(全局光和位置光)
float globalAmbient[4] = { 0.7f, 0.7f, 0.7f, 1.0f };
float lightAmbient[4] = { 0.0f, 0.0f, 0.0f, 1.0f };
float lightDiffuse[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
float lightSpecular[4] = { 1.0f, 1.0f, 1.0f, 1.0f };

// 金字塔的黄金材质
float* goldMatAmb = Utils::goldAmbient();
float* goldMatDif = Utils::goldDiffuse();
float* goldMatSpe = Utils::goldSpecular();
float goldMatShi = Utils::goldShininess();

// 环面的青铜材质
float* bronzeMatAmb = Utils::bronzeAmbient();
float* bronzeMatDif = Utils::bronzeDiffuse();
float* bronzeMatSpe = Utils::bronzeSpecular();
float bronzeMatShi = Utils::bronzeShininess();

// 在display()中将光照传入着色器的变量
float curAmb[4], curDif[4], curSpe[4], matAmb[4], matDif[4], matSpe[4];
float curShi, matShi;

// 阴影相关变量
int screenSizeX, screenSizeY;
GLuint shadowTex, shadowBuffer;
glm::mat4 lightVmatrix;
glm::mat4 lightPmatrix;
glm::mat4 shadowMVP1;
glm::mat4 shadowMVP2;
glm::mat4 b;

// 这里定义类型为mat4的光源观察矩阵与相机观察矩阵的矩阵变换(mMat, vMat等)
// 其他在display中所使用的变量也在此定义
   . . .
int main(void) {
   // 与前例相同，无改动
}

// init()函数依然执行调用以编译着色器并初始化物体
// 同时它也调用setupShadowBuffers()函数以初始化阴影贴图相关缓冲区
// 最后，它构造B矩阵以进行从光照空间到纹理空间的转换

void init(GLFWwindow* window) {

   renderingProgram1 = Utils::createShaderProgram("./vert1Shader.glsl", "./frag1Shader.glsl");
   renderingProgram2 = Utils::createShaderProgram("./vert2Shader.glsl", "./frag2Shader.glsl");

   setupVertices();
   setupShadowBuffers();

   b = glm::mat4(
       0.5f, 0.0f, 0.0f, 0.0f,
       0.0f, 0.5f, 0.0f, 0.0f,
```

```
        0.0f, 0.0f, 0.5f, 0.0f,
        0.5f, 0.5f, 0.5f, 1.0f);
}

void setupShadowBuffers(GLFWwindow* window) {
    glfwGetFramebufferSize(window, &width, &height);
    screenSizeX = width;
    screenSizeY = height;

    // 创建自定义帧缓冲区
    glGenFramebuffers(1, &shadowBuffer);

    // 创建阴影纹理并让它存储深度信息
    // 这些步骤与程序 5.2 中相似
    glGenTextures(1, &shadowTex);
    glBindTexture(GL_TEXTURE_2D, shadowTex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
        screenSizeX, screenSizeY, 0, GL_DEPTH_COMPONENT, GL_FLOAT, 0);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
}

void setupVertices(void) {
    // 与之前的例子相同。这个函数用来创建 VAO 和 VBO
    // 之后将环面及金字塔的顶点与法向量读入缓冲区
}

// display()函数分别管理第 1 轮需要使用的自定义帧缓冲区
// 以及第 2 轮需要使用的阴影纹理初始化过程。阴影相关新功能已高亮

void display(GLFWwindow* window, double currentTime) {
    glClear(GL_COLOR_BUFFER_BIT);
    glClear(GL_DEPTH_BUFFER_BIT);

    // 从光源视角初始化视觉矩阵以及透视矩阵，以便在第 1 轮中使用
    lightVmatrix = glm::lookAt(currentLightPos, origin, up); // 从光源到原点的矩阵
    lightPmatrix = glm::perspective(toRadians(60.0f), aspect, 0.1f, 1000.0f);

    // 使用自定义帧缓冲区，将阴影纹理附着到其上
    glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer);
    glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, shadowTex, 0);

    // 关闭绘制颜色，同时开启深度计算
    glDrawBuffer(GL_NONE);
    glEnable(GL_DEPTH_TEST);

    passOne();

    // 使用显示缓冲区，并重新开启绘制
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, shadowTex);
    glDrawBuffer(GL_FRONT);                 // 重新开启绘制颜色

    passTwo();
}

// 接下来是第 1 轮和第 2 轮的代码
```

```
// 这些代码和之前的大体相同
// 与阴影相关的新增代码已高亮

void passOne(void) {

   // renderingProgram1 包含了第 1 轮中的顶点着色器和片段着色器
   glUseProgram(renderingProgram1);
   . . .
   // 接下来的代码段通过从光源角度渲染环面获得深度缓冲区

   mMat = glm::translate(glm::mat4(1.0f), torusLoc);
   // 轻微旋转以便查看
   mMat = glm::rotate(mMat, toRadians(25.0f), glm::vec3(1.0f, 0.0f, 0.0f));

   // 我们从光源角度绘制，因此使用光源的 P、V 矩阵
   shadowMVP1 = lightPmatrix * lightVmatrix * mMat;
   sLoc = glGetUniformLocation(renderingProgram1, "shadowMVP");
   glUniformMatrix4fv(sLoc, 1, GL_FALSE, glm::value_ptr(shadowMVP1));

   // 在第 1 轮中我们只需要环面的顶点缓冲区，而不需要它的纹理或法向量
   glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
   glEnableVertexAttribArray(0);

   glClear(GL_DEPTH_BUFFER_BIT);
   glEnable(GL_CULL_FACE);
   glFrontFace(GL_CCW);
   glEnable(GL_DEPTH_TEST);
   glDepthFunc(GL_LEQUAL);

   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[4]);        // vbo[4] 包含环面索引
   glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);

   // 对金字塔做同样的处理(但不清除 GL_DEPTH_BUFFER_BIT)
   // 金字塔没有索引，因此我们使用 glDrawArrays()而非 glDrawElements()
   . . .
   glDrawArrays(GL_TRIANGLES, 0, numPyramidVertices);
}

void passTwo(void) {
   glUseProgram(renderingProgram2);        // 第 2 轮顶点着色器和片段着色器

   // 绘制环面，这次我们需要加入光照、材质、法向量等
   // 同时我们需要为相机空间以及光照空间都提供 MVP 变换
   mvLoc = glGetUniformLocation(renderingProgram2, "mv_matrix");
   projLoc = glGetUniformLocation(renderingProgram2, "proj_matrix");
   nLoc = glGetUniformLocation(renderingProgram2, "norm_matrix");
   sLoc = glGetUniformLocation(renderingProgram2, "shadowMVP");

   // 环面是黄铜材质
   curAmb[0] = bronzeMatAmb[0];  curAmb[1] = bronzeMatAmb[1];  curAmb[2] = bronzeMatAmb[2];
   curDif[0] = bronzeMatDif[0];  curDif[1] = bronzeMatDif[1];  curDif[2] = bronzeMatDif[2];
   curSpe[0] = bronzeMatSpe[0];  curSpe[1] = bronzeMatSpe[1];  curSpe[2] = bronzeMatSpe[2];
   curShi = bronzeMatShi;

   vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraLoc.x, -cameraLoc.y, -cameraLoc.z));

   currentLightPos = glm::vec3(lightLoc);
   installLights(renderingProgram2, vMat);
```

```
    mMat = glm::translate(glm::mat4(1.0f), torusLoc);
    // 轻微旋转以便查看
    mMat = glm::rotate(mMat, toRadians(25.0f), glm::vec3(1.0f, 0.0f, 0.0f));

    // 构建相机视角环面的 MV 矩阵
    mvMat = vMat * mMat;
    invTrMat = glm::transpose(glm::inverse(mvMat));

    // 构建光源视角环面的 MV 矩阵
    shadowMVP2 = b * lightPmatrix * lightVmatrix * mMat;

    // 将 MV 以及 PROJ 矩阵传入相应的统一变量
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    glUniformMatrix4fv(nLoc, 1, GL_FALSE, glm::value_ptr(invTrMat));
    glUniformMatrix4fv(sLoc, 1, GL_FALSE, glm::value_ptr(shadowMVP2));

    // 初始化环面顶点和法向量缓冲区()
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);          // 环面顶点
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);          // 环面法向量
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[4]);          // vbo[4]包含环面索引
    glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
    . . .
    // 对黄金金字塔重复同样步骤
}
```

程序 8.1 展示了与之前详述过的第 1 轮、第 2 轮着色器交互部分的 C++ / OpenGL 应用程序。之前已经展示过的模块，如读取着色器、编译着色器、构建模型及相关缓冲区、在着色器中初始化位置光源 ADS 特性以及进行透视矩阵和 LookAt 矩阵计算等，这些模块同之前一样。

8.6 阴影贴图的伪影

虽然我们已经实现了为场景添加阴影的所有基本要求，但运行程序 8.1 会产生错杂的结果，如图 8.11 所示。

好消息是我们的金字塔现在在环面上投下阴影！坏消息则是，这种成功伴随着严重的伪影。有许多波浪线覆盖在场景中的表面。这是阴影贴图的常见副作用，称为*阴影痤疮*（也称为*阴影斑块*，*shadow acne*）或*错误的自阴影*。

阴影痤疮是由深度测试期间的舍入误差引起的。在阴影纹理中查找深度信息时计算的纹理坐标通常与实际坐标不完全匹配。因此，从阴影纹理中查找到的深度值可能并非当前渲

染中像素的深度，而是相邻像素的深度。如果相邻像素在更远位置，则当前像素会被错误地显示为阴影。

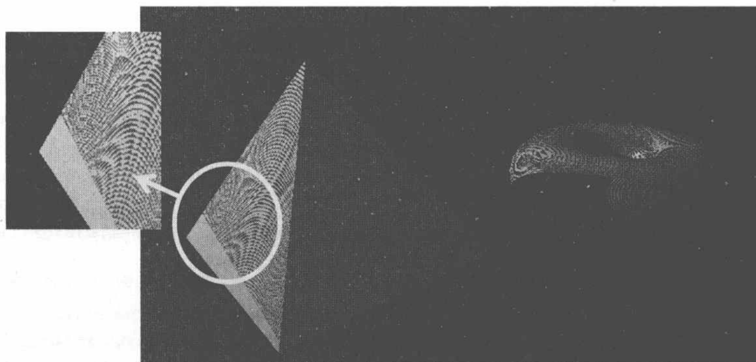


图 8.11 阴影的“痤疮”

阴影痤疮也会由纹理贴图和深度计算之间的精度差引起。这也可能导致舍入误差，并造成对像素是否处于阴影中的误判。

幸运的是，阴影痤疮很容易修复。由于阴影痤疮通常发生在没有阴影的表面上，这里有个简单的技巧，在第 1 轮中将每个像素稍微移向光源，之后在第 2 轮将它们移回原位。通常，这么做足以补偿各类舍入误差。在我们的实现中简单地在 display()函数中调用 glPolygonOffset()即可，如图 8.12 所示（突出显示部分）。

```
void display(GLFWwindow* window, double currentTime) {
  //像前面一样清除深度缓冲区和颜色缓冲区
  ...
  //像前面一样为相机和光源视角设置变换矩阵
  ...
  glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer);
  glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,shadowTex, 0);

  glDrawBuffer(GL_NONE);
  glEnable(GL_DEPTH_TEST);
  //减少阴影伪影
  glEnable(GL_POLYGON_OFFSET_FILL);
  glPolygonOffset(2.0f, 4.0f);

  passOne();

  glDisable(GL_POLYGON_OFFSET_FILL);

  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, shadowTex);
  glDrawBuffer(GL_FRONT);

  passTwo();
}
```

图 8.12 与阴影痤疮的战斗

将这几行代码添加到 display()函数，可以显著改善程序的输出，如图 8.13 所示。还要注意，随着伪影的消失，现在可以看到环面的内圆在其自身上显示了一个正确的小阴影。

虽然修复阴影痤疮很容易，但有时修复会引起新的伪影。在第 1 轮之前移动对象的“技巧”有时会导致在对象阴影中出现间隙。图 8.14 显示了一个这样的例子。这种伪影通常被称为“Peter Panning”，因为有时它会导致静止物体的阴影与物体底部分离的问题（从而使

物体的阴影部分与阴影的其余部分分离，让人想起詹姆斯·马修·巴利笔下的角色 Peter Pan[PP16]）。修复此伪影需要调整 glPolygonOffset()的参数。如果它们太小，就会出现阴影痤疮；如果太大，则会出现 Peter Panning。

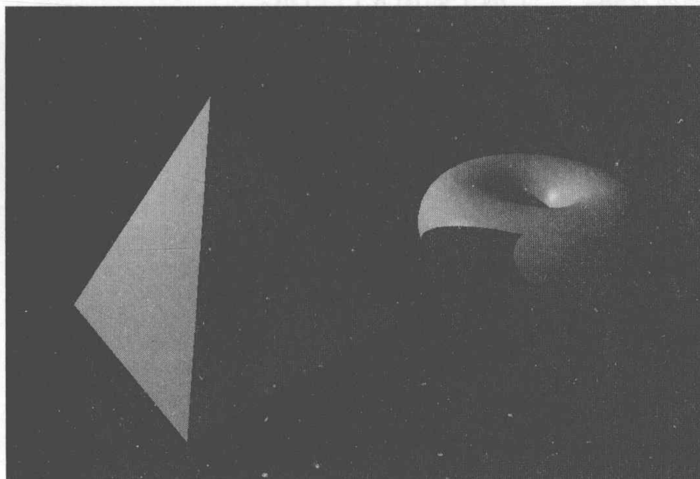


图 8.13　渲染带阴影的场景

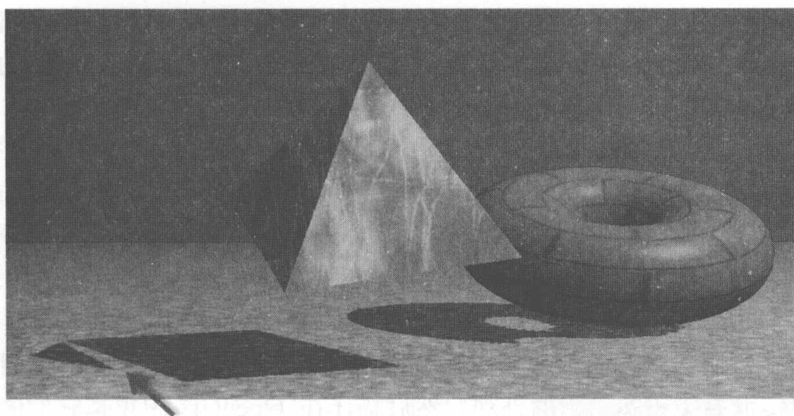


图 8.14　修复引起新的伪影

在实现阴影贴图时可能会发生许多其他伪影。如重复阴影，因为在第 1 轮（存入阴影缓冲区时）渲染的场景区域与第 2 轮中渲染的场景区域不同（来自不同的观察位置）。这种差异可能导致在第 2 轮中渲染的场景中，某些区域尝试使用范围[0…1]之外的特征坐标来访问阴影纹理。回想一下，在这种情况下默认行为是 GL_REPEAT，因此，这可能导致错误的重复阴影。

一种可能的解决方案是将以下代码行添加到 setupShadowBuffers()，将纹理换行模式设置为“夹紧到边缘”。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

这样纹理边缘以外的值会被限制为边缘处的值（而非重复）。注意，这种方法自身也有可能造成伪影，即当阴影纹理的边缘处存在阴影时，截取边缘可能产生延伸到场景边缘的

"阴影条"。

另一种常见错误是*锯齿状阴影边缘*。当投射的阴影明显大于阴影缓冲区可以准确表示的阴影时，就有可能出问题。通常，这取决于场景中物体和灯光的位置。尤其当光源在距离物体较远时，更容易发生。一个例子如图 8.15 所示。

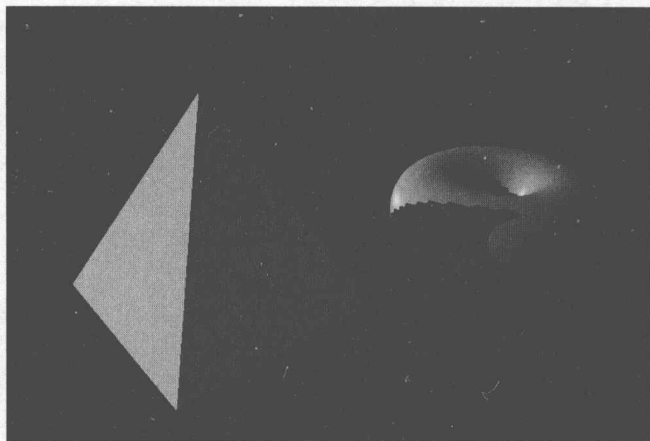


图 8.15　锯齿状阴影边缘

消除锯齿状阴影边缘就没有处理之前的伪影那么简单了。一种技术是在第 1 轮期间将光位置移动到更接近场景的位置，然后在第 2 轮放回原始位置。另一种常用的有效方法则是我们将在下面讨论的"柔和阴影"方法之一。

8.7 柔和阴影

目前我们所展示的阴影生成方法都仅限于生成*硬阴影*，即带锐边的阴影。但是，现实世界中出现的大多数阴影都是*柔和阴影*。它们的边缘都会发生不同程度的模糊。在本节中，我们将探讨现实世界中柔和阴影的外观，然后描述在 OpenGL 中模拟它们的常用算法。消除锯齿状阴影边缘并不像处理之前的伪影那么简单。

8.7.1 现实世界中的柔和阴影

柔和阴影的成因有很多，同时也有许多类型的柔和阴影。通常在自然界中产生柔和阴影原因是，真实世界的光源很少是点光源——它们常常是区域光源。另一个原因是材料和表面的缺陷积累，以及物体本身通过其自身的反射特性产生环境光的作用。

图 8.16 展示了物体向桌面投射柔和阴影的照片示例。注意，这不是计算机渲染的 3D 场景，而是真实的照片，是本书作者之一在家中拍摄的。

对于图 8.16 中的阴影，有两点需要注意。

- 离物体越远的阴影越"柔和"，离物体越近的阴影越"硬"。在对比物体腿附近的阴影与右边更宽的阴影时，这一点就很明显了。
- 距离物体越近的阴影显得越暗。

光源本身的维度会导致柔和阴影。如图 8.17 所示，光源上各处会投射出略微不同的阴影。各种阴影不同的区域称为半影（penumbra），包括阴影边缘的柔和区域。

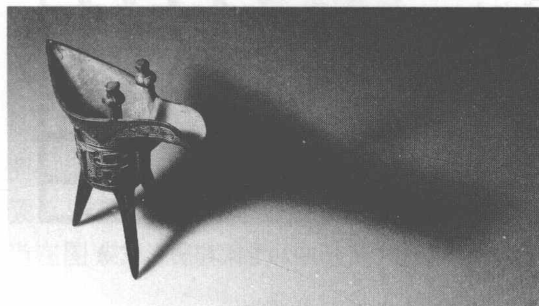


图 8.16 现实世界中的柔和阴影示例

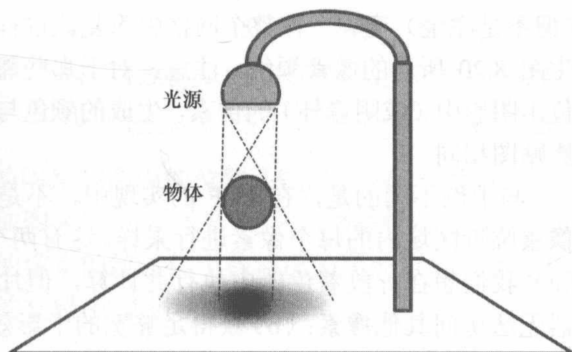


图 8.17 柔和阴影的半影效果

8.7.2 生成柔和阴影——百分比邻近滤波（PCF）

有多种方法可以用来模拟半影效果以在软件中生成柔和阴影。最简单也最常见的一种方法叫作百分比邻近滤波（Percentage Closer Filtering，PCF）。在 PCF 中，我们对单个点周围的几个位置的阴影纹理进行采样，以估计附近位置在阴影中的百分比。根据附近位置在阴影中的数量，对正在渲染的像素的光照分量进行修改。整个计算可以在片段着色器中完成，所以我们只需要对其中的代码进行修改。PCF 还可用于减少锯齿线伪影。

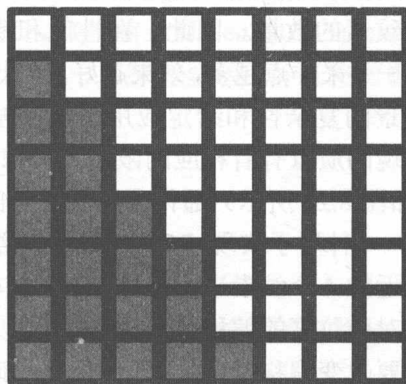


图 8.18 硬阴影渲染

在研究实际的 PCF 算法之前，我们先看一个类似的简单示例来展示 PCF 的目标。考虑图 8.18 中所示的输出片段（像素）集，其颜色由片段着色器计算。

假设深色像素处于阴影中，这是阴影贴图计算的结果。假设我们可以访问相邻的像素信息，而不是简单地如图所示渲染像素（即包括或不包括漫反射和镜面反射分量），这样我们就可以看到有多少相邻像素处于阴影中。例如，考虑图 8.19（见彩插）中以黄色突出显示的特定像素，根据图 8.18，该像素不在阴影中。

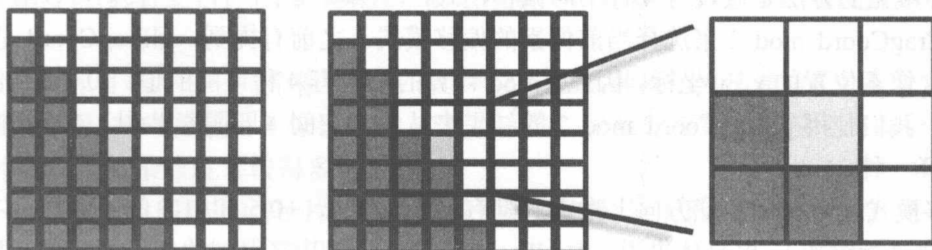


图 8.19 单像素 PCF 采样

在高亮像素的 9 个像素邻域中，3 个像素处于阴影中而 6 个像素处于阴影外。因此，渲染像素的颜色可以被计算为该像素处的环境光分量加上漫反射和镜面反射分量的 9/6，这样会使像素一定程度（但不是完全）变亮。在整个网格中重复此过程将会产生图 8.20 所示的像素颜色。注意，对于那些邻域完全位于阴影中（或阴影外）的像素，生成的颜色与标准阴影贴图相同。

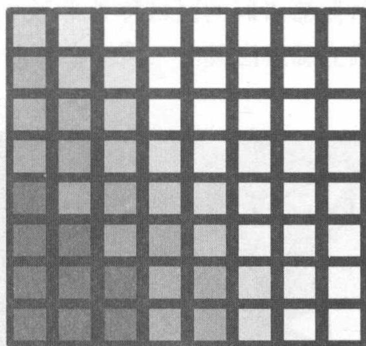


图 8.20 柔和阴影渲染

与上例不同的是，在 PCF 的实现中，不是对渲染像素临近区域内的每个像素进行采样。这有两个原因：(a) 我们想在片段着色器中执行此计算，但片段着色器无法访问其他像素；(b) 获得足够宽的半影效果（例如，10～20 像素宽）将需要为每个被渲染的像素采样数百个附近的像素。

PCF 解决了以下两个问题。首先，我们不试图访问附近的像素，而是在阴影贴图中对附近的纹素进行采样。片段着色器可以执行此操作，因为虽然它无法访问附近像素的值，但它可以访问整个阴影贴图。其次，为了获得足够宽的半影效果，需要对附近一定数量的阴影贴图纹素进行采样，每个采样的纹素都距离所渲染像素的纹素一定距离。

半影的宽度和采样点数可以根据场景和性能要求调整。例如，图 8.21 所示 PCF 生成的图像是，每个像素的亮度是通过对 64 个不同的纹素进行采样确定的，它们与像素的纹素距离各不相同。

柔和阴影的准确度或平滑度取决于所采样附近纹素的数量。因此，在性能和质量之间需要权衡——采样点越多，效果越好，但计算开销也越多。场景的复杂性和给定应用所需的帧率对于阴影可实现的质量有着相应的限制。每像素采样 64 个点（如图 8.21 所示）通常是不切实际的。

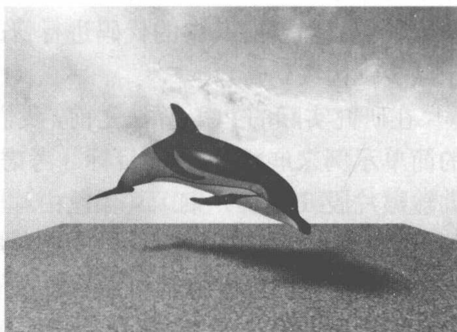


图 8.21 柔和阴影渲染——每像素 64 次采样

一种用于实现 PCF 的常见算法是对每个像素附近的 4 个纹素进行采样，其中样本通过指定从像素对应纹素的偏移量选择。对于每个像素，我们都需要改变偏移量，并用新的偏移量确定采样的 4 个纹素。用交错方式改变偏移量的方法被称为抖动，它旨在使得柔和阴影边界不会由于采样点不足看起来“结块”。

一种常见的方法是假设有 4 种不同偏移模式，每次取其中一种——我们可以通过计算像素的 glFragCoord mod 2 来选择当前像素的偏移模式。之前有提到，glFragCoord 是 vec2 类型，包含像素位置的 *x*、*y* 坐标。因此，mod 计算的结果有 4 种可能的值：(0,0)、(0,1)、(1,0)或(1,1)。我们使用 glFragCoord mod 2 的结果来从纹素空间（即阴影贴图）4 种不同偏移模式中选择一种。

偏移模式通常在 *x* 和 *y* 方向上指定，具有−1.5，−0.5，+0.5 和+1.5 的不同组合（也可以根据需要进行缩放）。更具体来说，由 glFragCoord mod 2 计算得到的每种情况的 4 种常用偏移模式是：

偏移模式(0,0)	偏移模式(0,1)	偏移模式(1,0)	偏移模式(1,1)
采样点：	采样点：	采样点：	采样点：
$(s_x-1.5, s_y+1.5)$	$(s_x-1.5, s_y+0.5)$	$(s_x-0.5, s_y+1.5)$	$(s_x-0.5, s_y+0.5)$
$(s_x-1.5, s_y-0.5)$	$(s_x-1.5, s_y-1.5)$	$(s_x-0.5, s_y-0.5)$	$(s_x-0.5, s_y-1.5)$
$(s_x+0.5, s_y+1.5)$	$(s_x+0.5, s_y+0.5)$	$(s_x+1.5, s_y+1.5)$	$(s_x+1.5, s_y+0.5)$
$(s_x+0.5, s_y-0.5)$	$(s_x+0.5, s_y-1.5)$	$(s_x+1.5, s_y-0.5)$	$(s_x+1.5, s_y-1.5)$

S_x和S_y指的是与正在渲染的像素相对应的阴影贴图中的位置（S_x，S_y），在本章的代码示例中标识为 shadow_coord。这 4 种偏移模式如图 8.22 所示（见彩插），每种情况都以不同的颜色显示。在每种情况下，对应于正被渲染的像素的纹素位于该情况的图的原点。请注意，当在图 8.23（见彩插）中一起显示时，偏移的交错/抖动很明显。

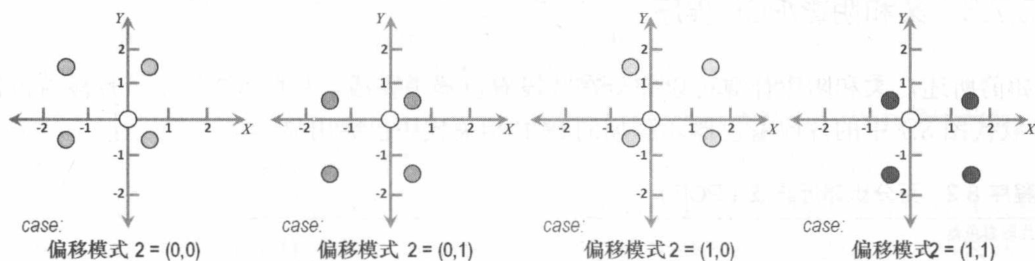


图 8.22 抖动的 4 像素 PCF 采样示例

让我们来针对特定像素看看整个计算过程。假设正在渲染的像素位于 glFragCoord = (48,13)。首先我们确定像素在阴影贴图的 4 个采样点。为此，我们将计算 vec2(48,13) mod 2，等于（0,1）。因此我们选择（0,1）所对应的偏移，在图 8.22 中以绿色显示，并且在阴影贴图对相应的点进行采样（假设没有指定偏移的缩放量），得到：

- (shadow_coord.x–1.5, shadow_coord.y+0.5)
- (shadow_coord.x–1.5, shadow_coord.y–1.5)
- (shadow_coord.x+0.5, shadow_coord.y+0.5)
- (shadow_coord.x+0.5, shadow_coord.y–1.5)

（回想一下，shadow_coord 是阴影贴图中与正在渲染的像素相对应的纹素的位置——在图 8.22 和图 8.23 中显示为白色圆圈）。

接下来，对我们选取的这 4 个点分别调用 textureProj()，在每种情况下都返回 0.0 或 1.0，具体取决于该采样点是否在阴影中。将 4 个结果相加并除以 4.0，就可以确定阴影中采样点的百分比。然后将此百分比用作乘数，确定渲染当前像素时要应用的漫反射和镜面反射分量。

尽管采样尺寸很小——每个像素只有 4 个样本——这种抖动方法通常可以产生好得惊人的柔和阴影。图 8.24 是使用 4 像素抖动 PCF 生成的。虽然它不如之前图 8.21 所示的 64 点采样版本好，但渲染速度要快得多。

在下一节中，我们对 GLSL 片段着色器进行编码，实现 4 采样抖动的 PCF 柔和阴影以及之前展示的 64 采样 PCF 柔和阴影。

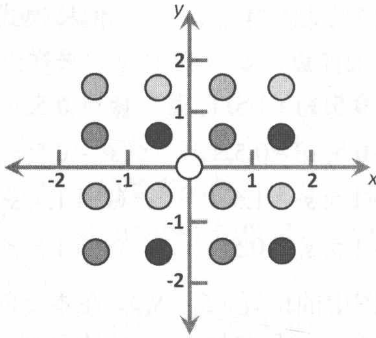


图 8.23 抖动的 4 像素 PCF 采样（4 种偏移模式）

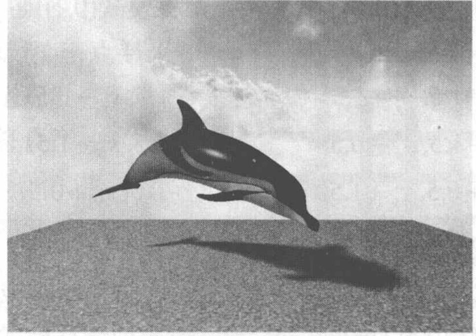


图 8.24 柔和阴影渲染——每像素 4 次采样，使用抖动

8.7.3 柔和阴影/PCF 程序

如前所述，柔和阴影计算可以完全在片段着色器中完成。程序 8.2 展示了片段着色器代码，取代图 8.7 中的片段着色器。添加的 PCF 相关代码已突出显示。

程序 8.2 百分比邻近滤波（PCF）

片段着色器

```
#version 430
// 所有变量定义未改动
. . .

// 从 shadow_coord 返回距离（x, y）处的纹素的阴影深度值
// shadow_coord 是阴影贴图中与正在渲染的当前像素相对应的位置

float lookup(float ox, float oy)
{ float t = textureProj(shadowTex,
    shadow_coord + vec4(ox * 0.001 * shadow_coord.w, oy * 0.001 * shadow_coord.w,
    -0.01, 0.0)); //第三个参数（-0.01）是用于消除阴影痤疮的偏移量
  return t;
}

void main(void)
{ float shadowFactor = 0.0;
    vec3 L = normalize(vLightDir);
    vec3 N = normalize(vNormal);
    vec3 V = normalize(-vVertPos);
    vec3 H = normalize(vHalfVec);

    // -----此部分生成一个 4 采样抖动的柔和阴影
    float swidth = 2.5;        //可调整的阴影扩散量
    // 根据 glFragCoord mod 2 生成 4 采样模式中的一个
    vec2 offset = mod(floor(gl_FragCoord.xy), 2.0) * swidth;
    shadowFactor += lookup(-1.5*swidth + offset.x, 1.5*swidth - offset.y);
    shadowFactor += lookup(-1.5*swidth + offset.x, -0.5*swidth - offset.y);
    shadowFactor += lookup( 0.5*swidth + offset.x, 1.5*swidth - offset.y);
    shadowFactor += lookup( 0.5*swidth + offset.x, -0.5*swidth - offset.y);
    shadowFactor = shadowFactor / 4.0; // shadowFactor 是 4 个采样点的平均值

    // ----- 取消本节注释以生成 64 采样的高分辨率柔和阴影
    // float swidth = 2.5;        // 可调整的阴影扩散量
```

```
    // float endp = swidth*3.0 +swidth/2.0;
    // for (float m=-endp ; m<=endp ; m=m+swidth)
    // {   for (float n=-endp ; n<=endp ; n=n+swidth)
    //     {            shadowFactor += lookup(m,n);
    // } }
    // shadowFactor = shadowFactor / 64.0;

    vec4 shadowColor = globalAmbient * material.ambient + light.ambient * material.ambient;
    vec4 lightedColor = light.diffuse * material.diffuse * max(dot(L,N),0.0)
                   + light.specular * material.specular
                   * pow(max(dot(H,N),0.0),material.shininess*3.0);

    fragColor = vec4((shadowColor.xyz + shadowFactor*(lightedColor.xyz)),1.0);
}
```

程序 8.2 中展示的片段着色器包含 4 采样和 64 采样的 PCF 柔和阴影的代码。为了更方便进行采样，我们需要定义 lookup()函数。在 lookup()函数中调用 GLSL 函数 textureProj()，从而在阴影纹理中以指定偏移量(*ox*, *oy*)进行查找。偏移量需要乘以 1 / *windowsize*，这里我们简单地假设窗口大小为 1 000 像素×1 000 像素，将乘数硬编码为 0.001。①

4 样本抖动的计算代码在 main()函数中高亮显示，其实现遵循上一节中描述的算法。同时添加了一个比例因子 swidth，用于调整阴影边缘的“柔和”区域的大小。

64 采样代码以注释形式出现在后面。可以通过取消 64 采样代码注释并注释 4 采样代码以使用 64 采样。在 64 采样代码中，swidth 比例因子用作嵌套循环中的步长，其采样距离正被渲染的像素的不同距离处的点。例如，当使用代码中的 swidth 值(2.5)时，程序将沿着每个轴在两个方向上以 1.25、3.75、6.25 和 8.25 的距离选择采样点——然后根据窗口大小进行缩放（如前所述）并用作纹理坐标采样阴影纹理。在这么多采样的情况下，通常不需要使用抖动来获得更好的结果。

图 8.25 展示了我们运行的环面/金字塔阴影贴图示例，它将 PCF 柔和阴影与程序 8.2 中的片段着色器相结合，分别使用了 4 采样和 64 采样的方法。swidth 的选值取决于场景；对于环面/金字塔示例，它的值为 2.5，而对于之前的图 8.21 中显示的海豚示例，swidth 的值为 8.0。

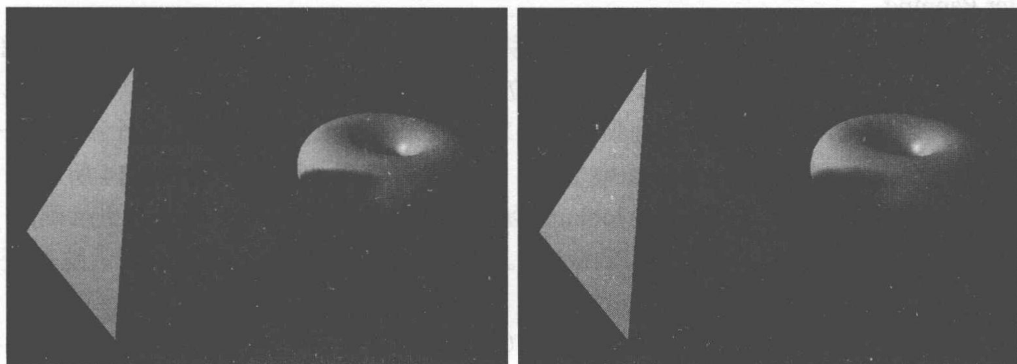


图 8.25 PCF 柔和阴影渲染——每像素 4 次采样，抖动（左）；每像素 64 次采样，不抖动（右）

① 我们还需要将偏移乘以阴影坐标的 *w* 分量，因为 OpenGL 在纹理查找期间会自动将输入坐标除以 *w*。迄今为止我们一直忽略了这种称为透视分割的操作，因此必须在这里说明。有关透视分割的更多信息，请参阅参考资料[LO12]。

补充说明

在本章中，我们仅给出了3D图形中阴影世界的最基本介绍。在更复杂的场景中，即便使用本章提供的基础阴影贴图方法，也可能需要进行进一步的研究。

例如，当场景中的某些对象拥有纹理的情况下，添加阴影时必须确保片段着色器正确区分阴影纹理和其他纹理。一种简单的方法是将它们绑定到不同的纹理单元，例如：

```
layout (binding = 0) uniform sampler2DShadow shTex;
layout (binding = 1) uniform sampler2D otherTexture;
```

然后，C++ / OpenGL应用程序可以通过它们的绑定值来引用两个采样器。

当场景使用多个灯光时，则需要多个阴影纹理——每个光源需要一个阴影纹理。此外，每个光源都需要单独执行第1轮渲染，并在第2轮渲染中合并结果。

尽管我们在阴影贴图的每个阶段都使用了透视投影，但值得注意的是，当光源是远距离光源和定向光源而非我们使用的位置光时，正射投影通常才是首选。

生成真实的阴影在计算机图形学中仍然是一个活跃而又复杂的领域，其中提出的许多技术超出了本书的范畴。我们鼓励对更多细节感兴趣的读者研究更专业的资源，如[ES12], [GP10]和[MI16]。

8.7.3小节包含一个GLSL函数的例子（除了“main”）。与在C语言中一样，必须在调用它们之前（或“上方”）定义函数，否则必须提供前向声明。在该示例中则不需要前向声明，因为函数定义在调用代码上方。

习题

8.1 在程序8.1中，尝试在不同设置下使用glPolygonOffset()，并观察对像的伪影效果，如Peter Panning。

8.2 （项目）修改程序8.1，以便通过移动鼠标移动灯光，类似于练习7.1。你可能会注意到某些照明位置会出现阴影伪影，而其他位置则没有。

8.3 （项目）给程序8.1添加动画，使得对象或光源（或两者一起）自行移动——例如一个绕另一个旋转。如果向场景添加地平面，阴影效果将更加明显，如图8.14所示。

8.4 （项目）修改程序8.2，将lookup()函数中的硬编码值0.001替换为更准确的1.0 / shadowbufferwidth和1.0 / shadowbufferheight。观察在窗口大小变化的情况下，这种变化产生了何种程度的影响（或没有影响）。

8.5 （研究）更复杂的百分比邻近滤波（PCF）的实现会加入光和阴影与光和遮挡物之间的相对距离。通过光线靠近或远离遮挡物时（或当遮挡物靠近或远离阴影时），调整半影的大小，可以使柔和阴影更逼真。研究此功能现有的实现方法，并将其添加到程序8.2中。

参考资料

[AS14] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with WebGL*, 7th ed. (Pearson, 2014).

[BL88] J. Blinn, "Me and My (Fake) Shadow," *IEEE Computer Graphics and Applications* 8, no. 2 (1988).

[CR77] F. Crow, "Shadow Algorithms for Computer Graphics," Proceedings of SIGGRAPH '77 11, no. 2 (1977).

[ES12] E. Eisemann, M. Schwarz, U. Assarsson, and M. Wimmer, *Real-Time Shadows* (CRC Press, 2012).

[GP10] *GPU Pro* (series), ed. Wolfgang Engel (A. K. Peters, 2010–2016).

[KS16] J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9th ed. (Addison-Wesley, 2016).

[LO12] "Understanding OpenGL's Matrices," Learn OpenGL ES (2012), accessed October 2018.

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[MI16] "Common Techniques to Improve Shadow Depth Maps" (Microsoft Corp., 2016), accessed October 2018.

[PP16] *Peter Pan*, Wikipedia, accessed October 2018.



第 9 章　天空和背景

对于室外 3D 场景，通常可以通过在地平线上创造一些逼真的效果，来增强其真实感。当我们极目远眺，目光越过附近的建筑和森林，我们习惯于看到远处的大型物体，例如：云、群山或太阳（或夜空中的星星和月亮）。但是，将这些对象作为单个模型添加到场景中可能会产生高到无法承受的性能成本。**天空盒**或**天空穹顶**提供了有效且相对简单的方法，用来生成令人信服的地平线景观。

9.1　天空盒

天空盒的概念非常巧妙而又简单：

（1）实例化一个立方体对象；

（2）将立方体的纹理设置为所需的环境；

（3）将立方体围绕相机放置。

我们已经知道如何完成以上这些步骤。但还有少量其他细节需要注意。

- 如何为地平线制作纹理？

立方体有 6 个面，我们需要为这些面都添加纹理。一种方法是使用 6 个图像文件和 6 个纹理单元。另一种常见（且高效）的方式则是使用一个包含 6 个面的纹理的图像，如图 9.1 所示。

上例中的纹理立方体贴图，仅用一个纹理单元，就可以为 6 个面添加纹理的图像。立方体贴图的 6 个部分对应于立方体的顶部、底部、正面、背面和两侧。当贴图“包裹”在立方体周围时，对于立方体内的相机而言，它扮演了地平线的角色，如图 9.2 所示。

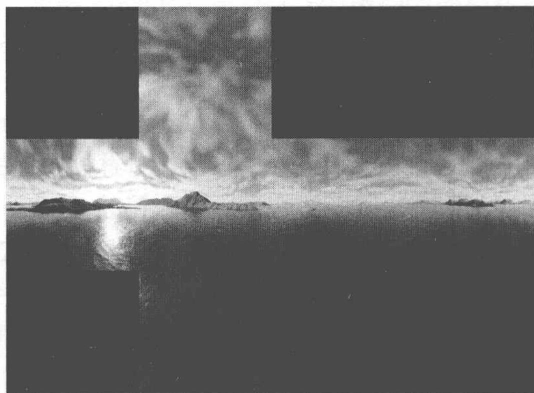


图 9.1　6 面天空盒纹理立方体贴图

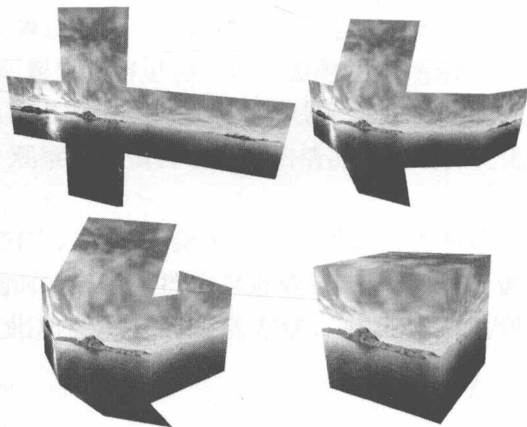


图 9.2　立方体贴图包裹相机

使用纹理立方体贴图为立方体添加纹理需要指定适当的纹理坐标。图 9.3 展示了纹理坐标的分布，这些坐标接着会分配给立方体的每个顶点。

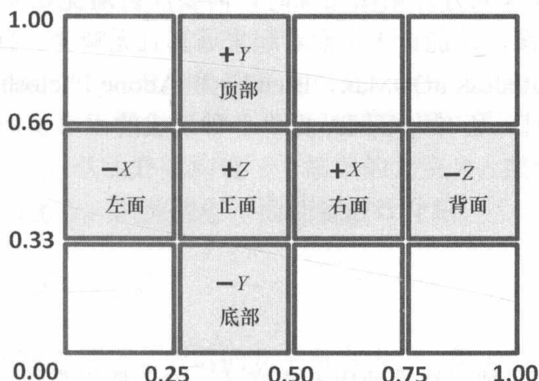


图 9.3 立方体贴图纹理坐标

- 如何让天空盒看起来“距离很远”？

构建天空盒的另一个重要因素是确保纹理的表现看起来像是远处的地平线。首先，人们可能会认为这需要构建巨大的天空盒。然而，事实证明这并不可取，因为巨大的天空盒会拉伸和扭曲纹理。相反，通过使用以下两个技巧，可以使天空盒显得巨大（从而感觉距离很远）：

（a）禁用深度测试并先渲染天空盒（在渲染场景中的其他对象时重新启用深度测试）；

（b）天空盒随相机移动（如果相机需要移动）。

通过在禁用深度测试的情况下先绘制天空盒，深度缓冲器的值仍将全设为 1.0（即最远距离）。因此，场景中的所有其他对象将被完全渲染，即天空盒不会阻挡任何其他对象。这样，无论天空盒的实际大小如何，会使天空盒的各面的位置看起来比其他物体都更远。而实际的天空盒立方体本身可以非常小，只要它在相机移动时随相机一起移动即可。图 9.4 展示了从天空盒内部查看简单的场景（实际上只有一个砖纹理环面）。

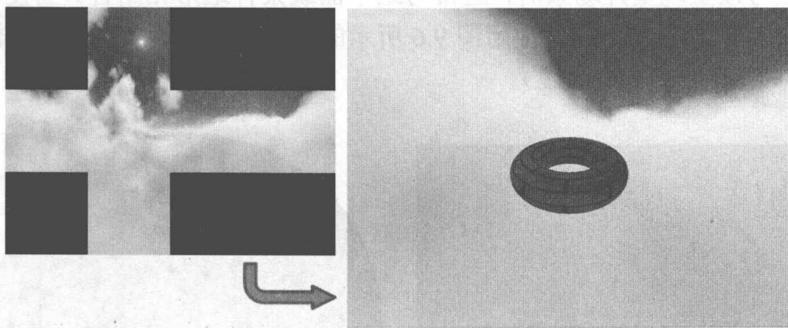


图 9.4 从天空盒内部查看场景

这里我们得益于对图 9.4 与之前图 9.2 和图 9.3 的关系的仔细研究。注意，场景中可见的天空盒部分是立方体贴图的最右侧部分。这是因为摄像机处于默认方向，面向−Z 方向，因此正在观察天空盒立方体的背面（如图 9.3 所示）。另请注意，立方体贴图的背面在场景中渲染时会呈水平反转状态；这是因为立方体贴图的“背面”部分已经折叠在相机周围，因

此看起来是经过侧向翻转的，如图 9.2 所示。

- 如何构建纹理立方体贴图？

从图稿或照片构建纹理立方体贴图图像时，需要注意避免在立方体面交汇点处的“接缝”，并创建正确的透视图，才能让天空盒看起来逼真且无畸变。有许多工具可以辅助达成这一目标：Terragen、Autodesk 3Ds Max、Blender 和 Adobe Photoshop 都有用于构建或处理立方体贴图的工具。同时，还有许多网站提供各种现成的立方体地图，既有付费的，也有免费的。

9.2 天空穹顶

建立地平线效果的另一种方法是使用**天空穹顶**。除了使用带纹理的球体（或半球体）代替带纹理的立方体，其基本思路与天空盒相同。与天空盒相同，我们首先渲染天空穹顶（禁用深度测试），并将摄像机保持在天空穹顶的中心位置（图 9.5 中的天空穹顶纹理是使用 Terragen [TE16]制作的）。

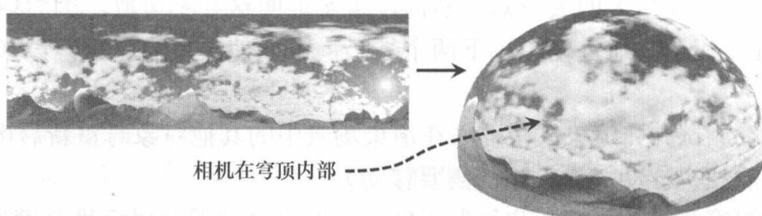


图 9.5 天空穹顶与其中的相机

天空穹顶相比天空盒有自己的优势。例如，它们不易受到畸变和接缝的影响（尽管在纹理图像中必须考虑极点处的球形畸变）。而天空穹顶的缺点之一则是球体或穹顶模型比立方体模型更复杂，天空穹顶有更多的顶点，其数量取决于期望的精度。

当使用天空穹顶呈现室外场景时，通常与地平面或某种地形相结合。当使用天空穹顶呈现宇宙中的场景（例如星空）时，使用图 9.6 所示的球体通常更为实际（为了清晰地使球体可视化，球体表面添加了一道虚线）。

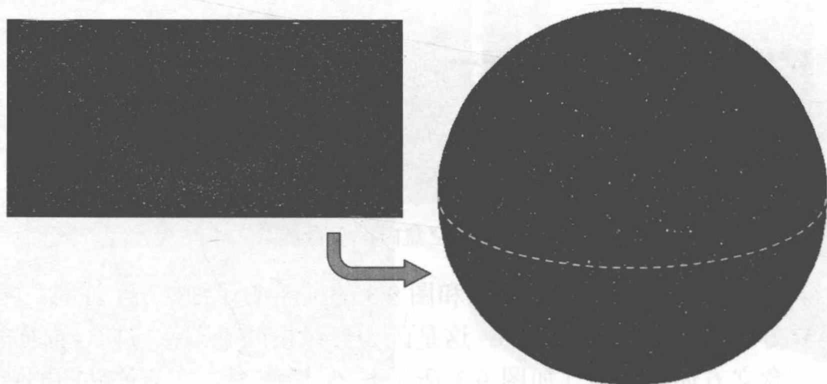


图 9.6 使用球体的星空天空穹顶（星图来自[BO01]）

9.3　实现天空盒

尽管天空穹顶有许多优点，天空盒仍然更为常见。OpenGL 对天空盒的支持也更好，在进行环境贴图时更方便（本章后面会介绍）。出于这些原因，我们将专注于天空盒的实现。

天空盒有两种实现方法：从头开始构建一个简单的天空盒；或使用 OpenGL 中的立方体贴图工具。它们有各自的优点，因此我们下面都会进行介绍。

9.3.1　从头开始构建天空盒

我们已经涵盖了构建简单天空盒所需的几乎所有内容。第 4 章介绍了立方体模型；分配纹理坐标已经在本章前面图 9.3 中进行了展示；使用 SOIL2 库读取纹理以及在 3D 空间中放置对象也都已经在之前的章节进行过讲解。这里，我们将看到如何简单地启用和禁用深度测试（只需要一行代码）。

程序 9.1 展示了简单天空盒的代码结构，场景中仅包含一个带纹理的环面。纹理坐标分配和启用/禁用深度测试的调用已突出显示。

程序 9.1　简单的天空盒

```
C++/OpenGL 应用程序

//所有变量声明，构造函数和 init()与之前相同
. . .
void display(GLFWwindow* window, double currentTime) {
    // 清除颜色缓冲区和深度缓冲区，并像之前一样创建投影视图矩阵和摄像机视图矩阵
    . . .
    glUseProgram(renderingProgram);

    // 准备首先绘制天空盒。M 矩阵将天空盒放置在摄像机位置
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cameraX, cameraY, cameraZ));

    // 构建 MODEL-VIEW 矩阵
    mvMat = vMat * mMat;

    // 如前，将 MV 和 PROJ 矩阵放入统一变量
    . . .

    // 设置包含顶点的缓冲区
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    // 设置包含纹理坐标的缓冲区
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(1,2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    //激活天空盒纹理
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, skyboxTexture);
```

```
   glEnable(GL_CULL_FACE);
   glFrontFace(GL_CCW);      // 立方体缠绕顺序是顺时针的，但我们从内部查看，因此使用逆时针缠绕顺序 GL_CCW

   glDisable(GL_DEPTH_TEST);
   glDrawArrays(GL_TRIANGLES, 0, 36);        // 在没有深度测试的情况下绘制天空盒
   glEnable(GL_DEPTH_TEST);

   //现在像之前一样绘制场景中的对象
   . . .
   glDrawElements( . . . );     //和之前的场景中的对象一样
}

void setupVertices(void) {
   // cube_vertices 定义与之前相同
   // 天空盒的立方体纹理坐标，如图 9.3 所示
   float cubeTextureCoord[72] = {
      1.00f, 0.66f, 1.00f, 0.33f, 0.75f, 0.33f,     // 背面右下角
      0.75f, 0.33f, 0.75f, 0.66f, 1.00f, 0.66f,     // 背面左上角
      0.75f, 0.33f, 0.50f, 0.33f, 0.75f, 0.66f,     // 右面右下角
      0.50f, 0.33f, 0.50f, 0.66f, 0.75f, 0.66f,     // 右面左上角
      0.50f, 0.33f, 0.25f, 0.33f, 0.50f, 0.66f,     // 正面右下角
      0.25f, 0.33f, 0.25f, 0.66f, 0.50f, 0.66f,     // 正面左上角
      0.25f, 0.33f, 0.00f, 0.33f, 0.25f, 0.66f,     // 左面右下角
      0.00f, 0.33f, 0.00f, 0.66f, 0.25f, 0.66f,     // 左面左上角
      0.25f, 0.33f, 0.50f, 0.33f, 0.50f, 0.00f,     // 下面右下角
      0.50f, 0.00f, 0.25f, 0.00f, 0.25f, 0.33f,     // 下面左上角
      0.25f, 1.00f, 0.50f, 1.00f, 0.50f, 0.66f,     // 上面右下角
      0.50f, 0.66f, 0.25f, 0.66f, 0.25f, 1.00f      // 上面左上角
   };
   //像往常一样为立方体和场景对象设置缓冲区
}
//用于加载着色器、纹理、main()等的模块，如前
```

标准纹理着色器现在用于场景中的所有对象，包括立方体贴图：

顶点着色器

```
#version 430
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 tex_coord;
out vec2 tc;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform sampler2D s;

void main(void)
{ tc = tex_coord;
  gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

片段着色器

```
#version 430
in vec2 tc;
out vec4 fragColor;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform sampler2D s;

void main(void)
{ fragColor = texture(s,tc);
}
```

程序 9.1 的输出如图 9.7 所示，包括两个不同立方体贴图纹理以及各自的渲染结果。

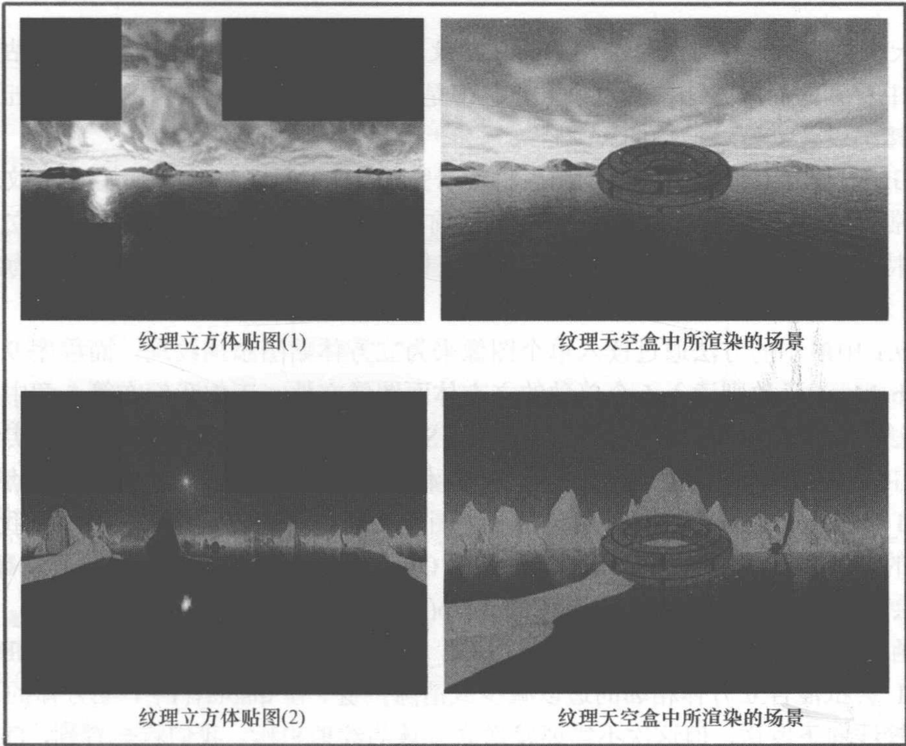


图 9.7 简单天空盒渲染结果

如前所述，天空盒容易受到图像畸变和接缝的影响。接缝指两个纹理图像接触的地方（比如沿着立方体的边缘）有时出现的可见线条。图 9.8 展示了一个图像上半部分出现接缝的示例，它是运行程序 9.1 时出现的伪影。为了避免接缝，需要仔细构建立方体贴图图像，并分配精确的纹理坐标。有一些工具可以用来沿图像边缘减少接缝（例如[GI16]），不过这个主题超出了本书的范围。

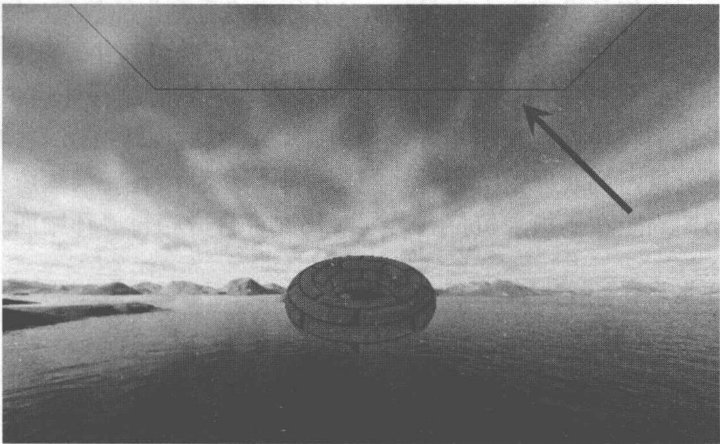


图 9.8 天空盒"接缝"伪影

9.3.2 使用 OpenGL 立方体贴图

构建天空盒的另一种方法是使用 OpenGL 纹理立方体贴图。OpenGL 立方体贴图比我们在上一节中看到的简单方法稍微复杂一点。但是，使用 OpenGL 立方体贴图有自己的优点，例如减少接缝以及支持环境贴图。

OpenGL 纹理立方体贴图类似于稍后将要研究的 3D 纹理，它们都使用 3 个纹理坐标访问——通常标记为（s, t, r）——而不是我们目前为止用到的两个。OpenGL 纹理立方体贴图的另一个特性是，其中的图像以纹理图像的左上角（而不是通常的左下角）作为纹理坐标（0, 0, 0），这通常是混乱产生的源头。

程序 9.1 中展示的方法通过读入单个图像来为立方体贴图添加纹理，而程序 9.2 中展示的 loadCubeMap()函数则读入 6 个单独的立方体面图像文件。正如我们在第 5 章中所学的，有许多方法可以读取纹理图像，我们选择使用 SOIL2 库。在这里，SOIL2 用于实例化和加载 OpenGL 立方体贴图也非常方便。我们先找到需要读入的文件，然后调用 SOIL_load_OGL_cubemap()，其参数包括 6 个图像文件和一些其他参数，类似于我们在第 5 章中看到的 SOIL_load_OGL_texture()。在使用 OpenGL 立方体贴图时，无须垂直翻转纹理，OpenGL 会自动进行处理，注意，loadCubeMap()函数放在“Utils.cpp”文件中。

init()函数现在包含一个函数调用以启用 GL_TEXTURE_CUBE_MAP_SEAMLESS，它告诉 OpenGL 尝试混合立方体相邻的边以减少或消除接缝。在 display()中，立方体的顶点像以前一样沿管线向下发送，但这次不需要发送立方体的纹理坐标。我们将会看到，OpenGL 纹理立方体贴图通常使用立方体的顶点位置作为其纹理坐标。之后禁用深度测试并绘制立方体。然后为场景的其余部分重新启用深度测试。

完成后的 OpenGL 纹理立方体贴图使用了 int 类型的标识符进行引用。与阴影贴图时一样，通过将纹理包裹模式设置为“夹紧到边缘”，可以减少沿边框的伪影。在这种情况下，它还可以帮助进一步缩小接缝。请注意，这里需要为 3 个纹理坐标 s、t 和 r 都设置纹理包裹模式。

在片段着色器中使用名为 samplerCube 的特殊类型的采样器访问纹理。在纹理立方体贴图中，从采样器返回的值是沿着方向向量(s, t, r)从原点“看到”的纹素。因此，我们通常可以简单地使用传入的插值顶点位置作为纹理坐标。在顶点着色器中，我们将立方体顶点位置分配到输出纹理坐标属性中，以便在它们到达片段着色器时进行插值。另外需要注意，在顶点着色器中，我们将传入的视图矩阵转换为 3×3，然后再转换回 4×4。这个“技巧”有效地移除了平移分量，同时保留了旋转（回想一下，平移值在转换矩阵的第四列中）。这样，就将立方体贴图固定在了摄像机位置，同时仍允许合成相机“环顾四周”。

程序 9.2 OpenGL 立方体贴图天空盒

```
C++/OpenGL application

. . .
int brickTexture, skyboxTexture;
int renderingProgram, renderingProgramCubeMap;
. . .
```

```
void init(GLFWwindow* window) {
   renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");
   renderingProgramCubeMap = Utils::createShaderProgram("vertCShader.glsl", "fragCShader.glsl");

   setupVertices();

   brickTexture = Utils::loadTexture("brick1.jpg");        // 场景中的环面
   skyboxTexture = Utils::loadCubeMap("cubeMap");          // 包含天空盒纹理的文件夹
   glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
}

void display(GLFWwindow* window, double currentTime) {
   // 清除颜色缓冲区和深度缓冲区，并像之前一样创建投影视图矩阵和摄像机视图矩阵

   . . .
   // 准备首先绘制天空盒—注意，现在它的渲染程序不同了

   glUseProgram(renderingProgramCubeMap);
   // 将 P、V 矩阵传入相应的统一变量
   . . .
   // 初始化立方体的顶点缓冲区（这里不再需要纹理坐标缓冲区）
   glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
   glEnableVertexAttribArray(0);

   // 激活立方体贴图纹理
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);

   // 禁用深度测试，之后绘制立方体贴图
   glEnable(GL_CULL_FACE);
   glFrontFace(GL_CCW);
   glDisable(GL_DEPTH_TEST);
   glDrawArrays(GL_TRIANGLES, 0, 36);
   glEnable(GL_DEPTH_TEST);
   // 绘制场景其余内容
   . . .
}

GLuint Utils::loadCubeMap(const char *mapDir) {
   GLuint textureRef;

   // 假设 6 个纹理图像文件 xp、xn、yp、yn、zp、zn 都是 JPG 格式图像
   string xp = mapDir; xp = xp + "/xp.jpg";
   string xn = mapDir; xn = xn + "/xn.jpg";
   string yp = mapDir; yp = yp + "/yp.jpg";
   string yn = mapDir; yn = yn + "/yn.jpg";
   string zp = mapDir; zp = zp + "/zp.jpg";
   string zn = mapDir; zn = zn + "/zn.jpg";

   textureRef = SOIL_load_OGL_cubemap(xp.c_str(), xn.c_str(), yp.c_str(), yn.c_str(),
      zp.c_str(), zn.c_str(), SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS);

   if (textureRef == 0) cout << "didnt find cube map image file" << endl;

   glBindTexture(GL_TEXTURE_CUBE_MAP, textureRef);

   // 减少接缝
   glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
   glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

```
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

    return textureRef;
}
```

顶点着色器

```
#version 430
layout (location = 0) in vec3 position;
out vec3 tc;

uniform mat4 v_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform samplerCube samp;

void main(void)
{
    tc = position;                                  // 纹理坐标就是顶点坐标
    mat4 vrot_matrix = mat4(mat3(v_matrix));        // 从视图矩阵中删除平移
    gl_Position = proj_matrix * vrot_matrix * vec4(position, 1.0);
}
```

片段着色器

```
#version 430
in vec3 tc;
out vec4 fragColor;

uniform mat4 v_matrix;
uniform mat4 proj_matrix;
layout (binding = 0) uniform samplerCube samp;

void main(void)
{ fragColor = texture(samp,tc);
}
```

9.4 环境贴图

在照明和材质章节中，我们考虑了物体的“光泽”。然而，我们从未对非常闪亮的物体进行建模，例如镜子或铬制品。这些物体在有小范围镜面高光的同时，还能够反射出周围物体的镜像。当我们看向这些物品时，我们会看到房间里的其他东西，有时甚至会看到我们自己的倒影。ADS 照明模型并没有提供模拟这种效果的方法。

不过，纹理立方体贴图提供了一种相对简单的方法来模拟（至少部分模拟）反射表面。其诀窍是使用立方体贴图来构造反射对象本身。[①]如果想要做得看起来真实，则需要找我们从物体上看到的周围环境所对应的纹理坐标。

图 9.9 展示了使用视图向量和法向量组合计算反射向量的策略，之后，该反射向量会用来从立方体贴图中查找纹素。因此，反射向量可用来直接访问纹理立方体贴图。当立方体贴图用于上述功能时，称其为环境贴图。

我们在之前研究 Blinn-Phong 照明时计算过反射向量。除了我们现在使用反射向量从纹理贴图中查找值，这里的反射向量概念和之前类似。这种技术称为**环境贴图**或**反射贴图**。

① 同样的技巧也适用于通过对反光物体添加天空穹顶纹理图像来用天空穹顶替代天空盒的情况。

如果使用我们描述的第二种方法(在 9.3.2 小节中,使用 OpenGL GL_TEXTURE_CUBE_MAP)实现立方体贴图，那么 OpenGL 可以使用与之前为立方体添加纹理相同的方法来进行环境贴图查找。我们使用视图向量和曲面法向量计算视图向量对应的离开对象表面的反射向量。然后可以使用反射向量直接对纹理立方体贴图图像进行采样。查找过程由 OpenGL samplerCube 辅助实现；回忆上一节中，samplerCube 使用视图方向向量索引。因此，反射向量非常适用于查找所需的纹素。

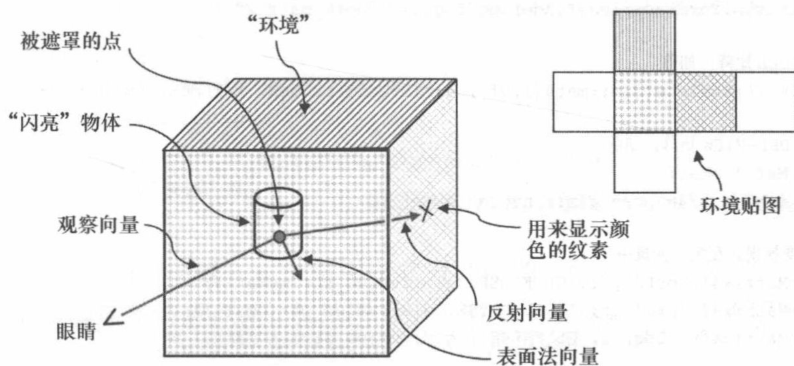


图 9.9 环境贴图总览

实现环境贴图需要添加相对少量的代码。程序 9.3 展示了 display()函数和 init()函数以及相关着色器中的更改，以使用环境贴图渲染“反射”环面。所有更改都已经高亮显示。值得注意的是，如果使用了 Blinn-Phong 照明，那么很多需要添加的代码可能已经存在了。真正新的代码部分在片段着色器中[在 main()函数中]。

乍一看程序 9.3 中突出显示的代码好像并不是新代码。实际上，在我们研究照明的时候，已经看到过几乎相同的代码。然而，在当前情况下，法向量和反射向量用于完全不同的目的。在之前的代码中，它们用于实现 ADS 照明模型。而在这里，它们用于计算环境贴图的纹理坐标。因此，我们将部分代码高亮，以便读者可以更轻松地追踪法向量和反射向量计算的使用，以实现这一新目的。

渲染的结果会显示使用了环境贴图的“铬制”环面，如图 9.10 所示（见彩插）。



图 9.10 用于创建反射环面的环境贴图示例

程序 9.3 环境贴图

```
void display(GLFWwindow* window, double currentTime) {
```

```
    // 用来绘制立方体贴图的代码未改变

    . . .
    // 所有修改都在绘制环面的部分

    glUseProgram(renderingProgram);

    // 矩阵变换的统一变量位置，包括法向量的变换
    mvLloc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
    nLoc = glGetUniformLocation(renderingProgram, "norm_matrix");

    // 构建MODEL矩阵，如前
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(torLocX, torLocY, torLocZ));

    // 构建MODEL-VIEW矩阵，如前
    mvMat = vMat * mMat;
    invTrMat = glm::transpose(glm::inverse(mvMat));

    // 法向量变换现在在统一变量中
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    glUniformMatrix4fv(nLoc, 1, GL_FALSE , glm::value_ptr(invTrMat));

    // 激活环面顶点缓冲区，如前
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    // 我们需要激活环面法向量缓冲区
    glBindBuffer(GL_ARRAY_BUFFER, vbo[2]);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(1);

    // 环面纹理现在是立方体贴图
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTexture);

    // 绘制环面的过程未做更改
    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_CULL_FACE);
    glFrontFace(GL_CCW);
    glDepthFunc(GL_LEQUAL);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[3]);
    glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
}
```

顶点着色器

```
#version 430
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
out vec3 varyingNormal;
out vec3 varyingVertPos;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
layout (binding = 0) uniform samplerCube tex_map;

void main(void)
```

```
{ varyingVertPos = (mv_matrix * vec4(position,1.0)).xyz;
  varyingNormal = (norm_matrix * vec4(normal,1.0)).xyz;
  gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

片段着色器

```
#version 430
in vec3 varyingNormal;
in vec3 varyingVertPos;
out vec4 fragColor;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform mat4 norm_matrix;
layout (binding = 0) uniform samplerCube tex_map;

void main(void)
{ vec3 r = -reflect(normalize(-varyingVertPos), normalize(varyingNormal));
  fragColor = texture(tex_map, r);
}
```

虽然该场景需要两组着色器——一组用于立方体贴图，另一组用于环面——但是程序 9.3 中仅展示了用于绘制环面的着色器。这是因为用于渲染立方体贴图的着色器与程序 9.2 中的相同。通过对程序 9.2 的修改得到程序 9.3 的过程，总结如下。

在 init()函数中：

- 创建环面的法向量缓冲区[实际上在 setupVertices()中完成，由 init()调用]；
- 不再需要环面的纹理坐标缓冲区。

在 display()函数中：

- 创建用于变换法向量的矩阵（在第 7 章中称为“norm_matrix”）并将其连接到关联的统一变量；
- 激活环面法向量缓冲区；
- 激活纹理立方体贴图为环面的纹理（而非之前的“砖”纹理）。

在顶点着色器中：

- 将法向量和 norm_matrix 相加；
- 输出变换的顶点和法向量以备计算反射向量，与在照明和阴影章节中所做的相似。

在片段着色器中：

- 以与照明章节中相似的方式计算反射向量；
- 从纹理（现在是立方体贴图）检索输出颜色，使用反射向量而非纹理坐标进行查找。

图 9.10 中显示的渲染结果是一个很好的例子，展示了通过简单的技巧能够实现强大的幻觉。通过在对象上简单地绘制背景，我们使对象看起来有“金属质感”，而根本没有进行 ADS 材质建模。即使没有任何 ADS 照明被整合到场景中，这种技巧也能让人感觉光从物体反射出来。在这个例子中，我们甚至会感到在环面的左下方似乎有一个镜面高光，因为立方体贴图中包括太阳在水中反射的倒影。

补充说明

正如我们在第 5 章中第一次研究纹理时的情况一样，使用 SOIL2 使得构建立方体贴图和为立方体贴图添加纹理变得容易。同时它也可能会有一些副作用，即阻挡用户学习一些有

用的 OpenGL 细节内容。当然，用户也可以在没有 SOIL2 的情况下实例化并加载 OpenGL 立方体贴图。虽然该主题超出了本书的范围，但基本步骤如下：

（1）使用 C++工具读取 6 个图像文件（它们必须是正方形）；

（2）使用 glGenTextures()为立方体贴图创建纹理及其整型引用；

（3）调用 glBindTexture()，指定纹理的 ID 和 GL_TEXTURE_CUBE_MAP；

（4）使用 glTexStorage2D()指定立方体贴图的存储需求；

（5）调用 glTexImage2D()或 glTexSubImage2D()将图像分配给立方体的各个面。

更多有关在没有 SOIL2 的情况下创建 OpenGL 立方体贴图的详细信息，请浏览互联网上的一些相关教程[dV14], [GE16]。

如本章所述，环境贴图的主要限制之一是它只能构建反射立方体贴图内容的对象。在场景中渲染的其他对象并不会出现在使用贴图模拟反射的对象中。这种限制是否可以接受取决于场景的性质。如果场景中存在必须出现在镜面或铬制对象中的对象，则必须使用其他方法。一种常见的方法是使用模板缓冲区（在第 8 章中有提到），许多网络教程（例如[OV12]、[NE14]和[GR16]）中都有描述，不过它超出了本书的范围。

我们没有介绍天空穹顶的实现，虽然它们在某些方面可以说比天空盒更简单，并且不易受到失真的影响，甚至用它实现环境贴图也更简单——至少在数学上——但 OpenGL 对立方体贴图的支持常常使得天空盒更加实用。

在书后面部分涵盖的主题中，天空盒和天幕在概念上可以说是最简单的。然而，让它们看起来令人信服可能会耗费大量时间。我们只简要介绍了可能出现的一些问题（例如接缝），但根据使用的纹理图像文件，可能会出现其他问题，需要额外修复。尤其是在动画场景中或相机可以通过交互进行移动时。

我们还大致介绍了如何生成可用且令人信服的纹理立方体贴图图像。这方面有许多优秀的工具，其中最受欢迎的是 Terragen [TE16]。本章中的所有立方体贴图均由作者使用 Terragen 制作（图 9.6 中的星域图除外）。

习题

9.1 （项目）在程序 9.2 中，添加使用鼠标或键盘移动相机的功能。为此，你需要使用先前在习题 4.2 中开发的代码来构建视图矩阵。还要为前后移动以及绕各轴旋转相机的功能分配鼠标或键盘操作（你需要编写这些函数）。完成这些操作后，你应该能够在场景中“飞来飞去”，并能够注意到天空盒始终看起来保持在遥远的地平线上。

9.2 在 6 个立方体贴图图像文件上绘制标签，以确认实现中使用了正确的方向。例如，你可以在图像上绘制轴标签，如图 9.11 所示。

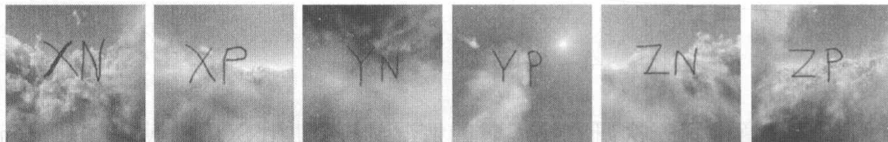


图 9.11 6 个立方体贴图

还可以使用“带标记的”立方体贴图来验证环境贴图环面中的反射是否正确呈现。

9.3 （项目）向程序 9.3 添加动画，以便场景中的一个（或多个）使用了环境贴图的对象旋转或翻转。当天空盒纹理在物体表面上移动时，物体的模拟反射性质会更明显。

9.4 （项目）修改程序 9.3，以使场景中的对象将环境贴图与纹理混合。在片段着色器中加权求和，如第 7 章中所述。

9.5 （研究和项目）了解使用 Terragen [TE16]创建简单立方体贴图的基础知识。通常需要（在 Terragen 中）制作具有所需地形和大气模式的“世界”，然后将 Terragen 的合成相机放置在前、后、右、左、顶部和底部以保存作为各视图的 6 个图像。在程序 9.2 和程序 9.3 中使用新生成的图像，观察它们作为立方体贴图和环境贴图呈现的外观。使用 Terragen 的免费版本足以进行此练习。

参考资料

[BO01] P. Bourke, “Representing Star Fields,” October 2018, accessed July 2016.

[dV14] J. de Vries, “Learn OpenGL – Cubemaps,” 2014, accessed October 2018.

[GE16] A. Gerdelan, “Cube Maps: Sky Boxes and Environment Mapping,” 2016, accessed October 2018.

[GI16] GNU Image Manipulation Program, accessed October 2018.

[GR16] OpenGL Resources, “Planar Reflections and Refractions Using the Stencil Buffer,” accessed October 2018.

[NE14] NeHeProductions, “Clipping and Reflections Using the Stencil Buffer,” 2014, accessed October 2018.

[OV12] A. Overvoorde, “Depth and Stencils,” 2012, accessed October 2018.

[TE16] Terragen, Planetside Software, LLC, accessed October 2018.



第 10 章 增强表面细节

假设我们想要对不规则表面的物体进行建模，例如橘子凹凸的表皮、葡萄干褶皱的表面或月球的陨石坑表面。我们该怎么做？到目前为止，我们已经学会了两种可能的方法：(a)我们可以对整个不规则表面进行建模，但这么做通常不切实际（一个有许多坑的表面需要大量的顶点）；(b)我们可以将不规则表面的纹理图图像应用于平滑的对象。第二种选择通常比较高效。但是，如果场景中有光源，当光源（或摄像机角度）移动时，我们很快就会发现物体使用了静态纹理渲染（以及物体表面是平滑的），因为纹理上的亮区和暗区不会像真正凹凸不平的表面那样，随着光源或摄像机移动而改变。

在本章中，我们将探讨几种与实现凹凸表面相关的方法，通过使用光照效果，即使在实际对象模型表面平滑的情况下，也能使对象看起来具有逼真的表面纹理。我们将首先观察**凹凸贴图和法线贴图**，当直接为对象添加微小表面细节会使得计算代价过高时，它们可以为场景中的对象增加相当程度的真实感。我们还将研究通过高度贴图实际扰乱光滑表面中顶点的方法，这对于生成地形（和其他一些用途）非常有用。

10.1 凹凸贴图

在第 7 章中，我们了解了表面法向量在创建令人信服的光照效果中是至关重要的。像素处的光强度主要由反射角确定，即需要考虑到光源位置、相机位置和像素处的法向量。因此，如果我们能找到生成相应法向量的方法，就可以避免生成与凹凸不平或褶皱表面相对应的顶点。

图 10.1 展示了对于单个“凸起”修改法向量的概念。

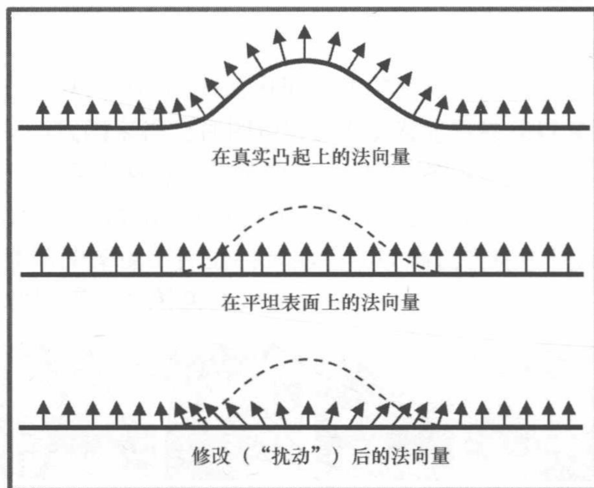


图 10.1 用于凹凸贴图的扰动法向量

因此，如果我们想让一个物体看起来好像有凹凸（或皱纹，陨石坑等），一种方法是计算当表面确实凹凸不平时其上的法向量。当场景点亮时，光照会让人产生我们所期望的幻觉。这是 Blinn 在 1978 年首次提出的[BL78]，随着在片段着色器拥有了可以对每个像素进行光照计算的能力，这种方法就变得切实可行了。

程序 10.1 中展示了顶点着色器和片段着色器的一个示例，这段程序会生成一个带有“高尔夫球”表面的环面，如图 10.2 所示。其代码几乎与我们之前在程序 7.2 中看到的相同。片段着色器中唯一显著的变化是——输入的已插值法向量（在原程序中名为“varyingNormal”）在这里变得凹凸不平了，其方法是对环面模型的原始（未变形）顶点的 *X*、*Y* 和 *Z* 轴应用正弦函数。请注意，这里需要顶点着色器将未经变换的顶点沿管线传递给片段着色器。

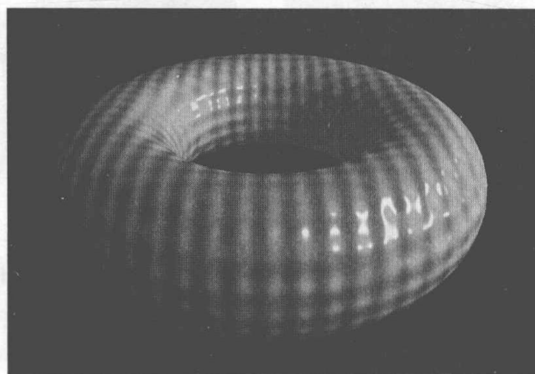


图 10.2 过程式凹凸贴图示例

以这种方式对法向量进行改变，即在运行时使用数学函数进行计算，称为**过程式凹凸贴图**。

程序 10.1 过程式凹凸贴图

顶点着色器

```
#version 430
// 与 Phong 着色相同，但添加此输出顶点属性
out vec3 originalVertex;

. . .
void main(void)
{ // 添加原始顶点，传递以进行插值
  originalVertex = vertPos;
  . . .
}
```

片段着色器

```
#version 430
// 与 Phong 着色相同，但添加此输入顶点属性
in vec3 originalVertex;

. . .
void main(void)
{ . . .
  // 添加如下代码以扰乱传入的法向量
  float a = 0.25;   // a 控制凸起的高度
  float b = 100.0;  // b 控制凸起的宽度
```

```
    float x = originalVertex.x;
    float y = originalVertex.y;
    float z = originalVertex.z;
    N.x = varyingNormal.x + a*sin(b*x);    // 使用正弦函数扰乱传入法向量
    N.y = varyingNormal.y + a*sin(b*y);
    N.z = varyingNormal.z + a*sin(b*z);
    N = normalize(N);
  // 光照计算以及输出的 fragColor（未更改）现在使用扰动过的法向量 N
  ...
}
```

10.2 法线贴图

凹凸贴图的一种替代方法是使用查找表来替换法向量。这样我们就可以在不依赖数学函数的情况下，对凸起进行构造，例如月球上的陨石坑所对应的凸起。一种使用查找表的常见方法叫作**法线贴图**。

为了理解法线贴图的工作原理，我们首先注意，向量通过 3 字节存储，*X*、*Y* 和 *Z* 分量各占 1 字节，就可以达到合理的精度。这样，我们就可以将法向量存储在彩色图像文件中，其中 *R*、*G* 和 *B* 分量分别对应于 *X*、*Y* 和 *Z*。图像中的 *RGB* 值以字节存储，通常被解释为[0…1]范围内的值，但是向量可以有正负值分量。如果我们将法向量分量限制在[−1…+1]范围内，那么在图像文件中将法向量 ***N*** 存储为像素的简单转换是：

$$R = (N_X + 1)/2$$
$$G = (N_Y + 1)/2$$
$$B = (N_Z + 1)/2$$

法线贴图使用一个图像文件（称为法线贴图），该图像文件包含在光照下所期望表面外观的法向量。在法线贴图中，向量相对于任意平面 *XY* 表示，其 *X* 和 *Y* 分量表示与“垂直”的偏差，其 *Z* 分量设置为 1，严格垂直于 *XY* 平面的向量（即没有偏差）将表示为（0, 0, 1），而不垂直的向量将具有非零的 *X* 和/或 *Y* 分量。我们需要使用上面的公式将值转换至 *RGB* 空间；例如，（0, 0, 1）将存储为（0.5, 0.5, 1），因为实际偏移的范围为[−1…+1]，而 *RGB* 值的范围为[0…1]。

我们可以通过纹理单元的另一种妙用来生成这样一幅法线贴图：我们在纹理单元中存储所需的法向量而非颜色。然后，在给定片段中，我们就可以使用采样器从法线贴图中查找值，接下来，我们将所得的值作为法向量，而非输出像素颜色（在纹理贴图中我们是这么做的）。

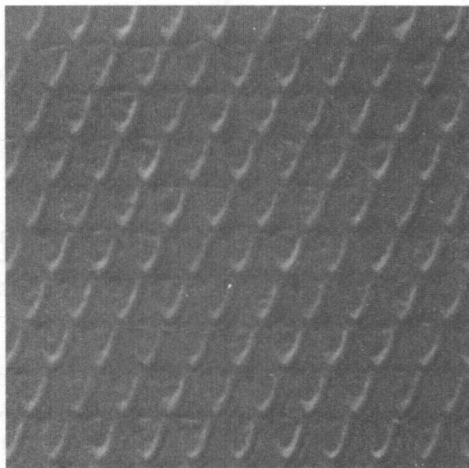


图 10.3 法线贴图图像文件示例[ME11]

图 10.3 展示了一个法线贴图图像文件的例子，通过将 GIMP 法线贴图插件[GI16]应用于 Luna [LU16]纹理而生成。法线贴图图像文件并不适合作为图像查看，我们展示这幅图就是为了指明这一点，法线贴图最终看起来基本都是蓝

色的。这是因为图像文件中每个像素的 *B* 值（蓝色值）都是 1（最大蓝色值），这会让它在作为图像时看起来是“蓝色的”。

图 10.4 展示了两个不同的法线贴图图像文件（它们都由 Luna [LU16]的纹理构建）以及在 Blinn-Phong 光照模型下将它们应用于球体的结果。

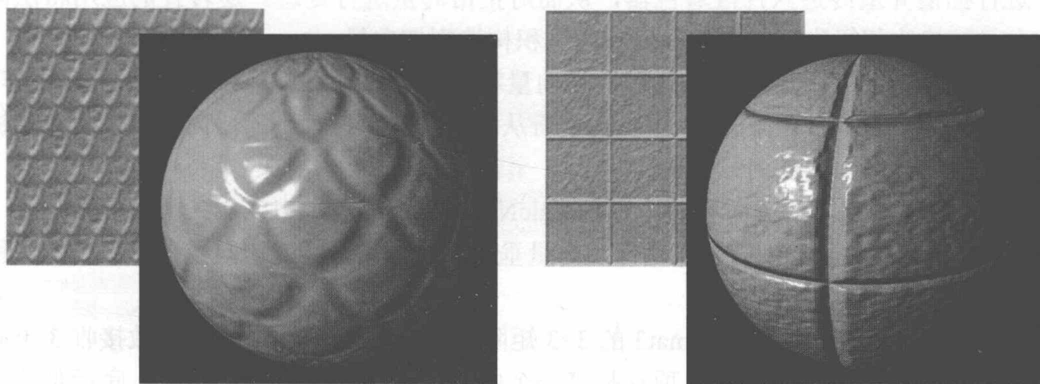


图 10.4 法线贴图示例

从法线贴图查找到的法向量不能直接使用，因为它们是相对于上述的任意 *XY* 平面定义的，并没有考虑它们在物体上的位置以及在相机空间中的方向。这个问题的解决策略是建立一个转换矩阵，用于将法向量转换为相机空间，如下所示。

在对象的每个顶点处，我们考虑与对象相切的平面。顶点处的物体的法向量垂直于该切面。我们在该切面中定义两个相互垂直的向量，同时也垂直于法向量，称为**切向量**和**副切向量**（有时称为副法向量）。构造我们期望的变换矩阵要求我们的模型包括每个顶点的切向量（可以通过计算切向量和法向量的叉积来构建副切向量）。如果模型中没有定义切向量，则需要通过计算得到它们。在球体的情况下，可以通过计算得到精确的切向量。以下是对程序 6.1 的修改：

```
. . .
for (int i=0; i<=prec; i++) {
    for (int j=0; j<=prec; j++) {
        float y = (float)cos(toRadians(180.0f - i*180.0f / prec));
        float x = -(float)cos(toRadians(j*360.0f / prec)) * (float)abs(cos(asin(y)));
        float z = (float)sin(toRadians(j*360.0f / prec)) * (float)abs(cos(asin(y)));
        vertices[i*(prec+1)+j] = glm::vec3(x, y, z);
        // 计算切向量
        if (((x==0) && (y==1) && (z==0)) || ((x==0) && (y==-1) && (z==0))) // 如果是北极或南极,
        {    tangent[i*(prec+1)+j] = glm::vec3(0.0f, 0.0f, -1.0f);         // 设置切向量为 -Z 轴
        }
        else                                // 否则，计算切向量
        {  tangent[i*(prec+1)+j] = glm::cross(glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(x,y,z));
        }
        . . . // 其余计算代码不变
    }
}
```

对于那些表面不可导以至于无法精确求解切向量的模型，其切向量可以通过近似得到，例如在构造（或加载）模型时，将每个顶点指向下一个顶点的向量作为切向量。请注意，

这种近似可能会导致切向量与顶点法向量不严格垂直。因此，如果要实现适用于各种模型的法线贴图，需要考虑这种可能性（我们的解决方案中对此进行了处理）。

切向量与顶点、纹理坐标以及法向量一样，是从缓冲区（VBO）传递到顶点着色器中的顶点属性。然后，顶点着色器通过应用 MV 矩阵的逆转置并将结果沿着流水线转发以由光栅器进行插值并最终进入片段着色器，从而对正常向量进行处理。逆转置的应用将法向量和切向量转换为相机空间，之后我们使用叉积构造副切向量。

一旦我们在相机空间中得到法向量、切向量和副切向量，就可以使用它们来构造矩阵（依其分量命名为“TBN”矩阵），该矩阵用于将从法线贴图中检索到的法向量转换为在相机空间中相对于物体表面的法向量。

在片段着色器中，新法向量的计算在 calcNewNormal()函数中完成。函数的第三行［包含 dot（tangent，normal）］的计算确保切向量垂直于法向量。新的切向量和法向量的叉积就是副切向量。

然后，我们创建一个类型为 mat3 的 3×3 矩阵，作为 TBN。mat3 构造函数接收 3 个向量作为参数，生成一个矩阵，其中顶行是第一个向量，中间行是第二个向量，底行是第三个向量（类似于从摄像机位置构建视图矩阵，见图 3.13）。

着色器使用片段的纹理坐标来提取与当前片段对应的法线贴图单元。着色器在提取时使用采样器变量“normMap”，并被绑定到纹理单元 0（注意：因此在 C++ / OpenGL 应用程序中必须将法线贴图图像附加到纹理单元 0）。因为需要将颜色分量从纹理中存储范围[0…1]转换为其原始范围[−1 … + 1]，我们将其乘以 2.0 再减去 1.0。

然后将 TBN 矩阵应用于所得法向量以产生当前像素的最终法向量。着色器的其余部分与用于 Phong 光照的片段着色器相同。片段着色器代码基于 Etay Meiri [ME11]的版本，如程序 10.2 所示。

制作法线贴图图像可以使用各种各样的工具。有的图像编辑工具就有制作法线贴图的功能，例如 GIMP [GI16]和 Photoshop [PH16]。它们通过分析图像中的边缘，推断凸起和凹陷，并产生相应的法线贴图。

图 10.5 显示了由 Hastings-Trew [HT16]基于 NASA 卫星数据创建的月面纹理图。其相应的法线贴图由 GIMP 法线贴图插件[GP16]，通过处理由 Hastings-Trew 创建的黑白版本月面纹理图生成。

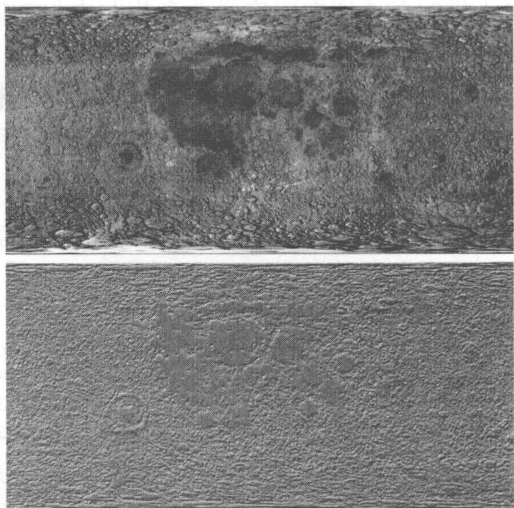


图 10.5 月球纹理（上）和法线贴图（下）

程序 10.2 法线贴图片段着色器

```
#version 430
in vec3 varyingLightDir;
in vec3 varyingVertPos;
in vec3 varyingNormal;
in vec3 varyingTangent;
```

```
in vec3 originalVertex;
in vec2 tc;
in vec3 varyingHalfVector;
out vec4 fragColor;

layout (binding=0) uniform sampler2D normMap;
// 其余统一变量同前
. . .
vec3 calcNewNormal()
{ vec3 normal = normalize(varyingNormal);
  vec3 tangent = normalize(varyingTangent);
  tangent = normalize(tangent - dot(tangent, normal) * normal); // 切向量垂直于法向量
  vec3 bitangent = cross(tangent, normal);
  mat3 tbn = mat3(tangent, bitangent, normal);          // 用来变换到相机空间的 TBN 矩阵
  vec3 retrievedNormal = texture(normMap,tc).xyz;
  retrievedNormal = retrievedNormal * 2.0 - 1.0;        // 从 RGB 空间转换
  vec3 newNormal = tbn * retrievedNormal;
  newNormal = normalize(newNormal);
  return newNormal;
}

void main(void)
{ // 正规化光照向量，法向量和视图向量
  vec3 L = normalize(varyingLightDir);
  vec3 V = normalize(-varyingVertPos);
  vec3 N = calcNewNormal();

  // 获得光照向量和曲面法向量之间的角度
  float cosTheta = dot(L,N);

  // 为 Blinn 优化计算半向量
  vec3 H = normalize(varyingHalfVector);

  // 视图向量和反射光向量之间的角度
  float cosPhi = dot(H,N);

  // 计算 ADS 贡献（每个像素）
  fragColor = globalAmbient * material.ambient
  + light.ambient * material.ambient
  + light.diffuse * material.diffuse * max(cosTheta,0.0)
  + light.specular * material.specular * pow(max(cosPhi,0.0), material.shininess*3.0);
}
```

图 10.6 展示了使用两种不同方式渲染的，用以表现月球表面的球体。图 10.6 左图中，球体使用了原始的纹理贴图；图 10.6 右图中，球体使用法线贴图的图像作为纹理（供参考）。它们都没有应用法线贴图。虽然左侧使用了纹理的“月球”非常逼真，但仔细观察可以发现，纹理图案很明显拍摄于阳光从左侧照亮月球的时候，因为其山脊的阴影投射到了右侧（在底部中心附近的火山口中最明显）。如果我们使用 Phong 着色为此场景添加光照，然后移动月球、相机或灯光来给场景添加动画，就会发现月球上的阴影不会如我们期望地改变。

此外，随着光源的移动（或相机移动），期望中会在山脊上出现许多镜面高光。但是图 10.6 左图使用了标准纹理的球体将只产生一个镜面高光，对应于光滑球体上所出现的高光，这看起来非常不现实。配合法线贴图可以显著提高这类对象在光照下的真实感。

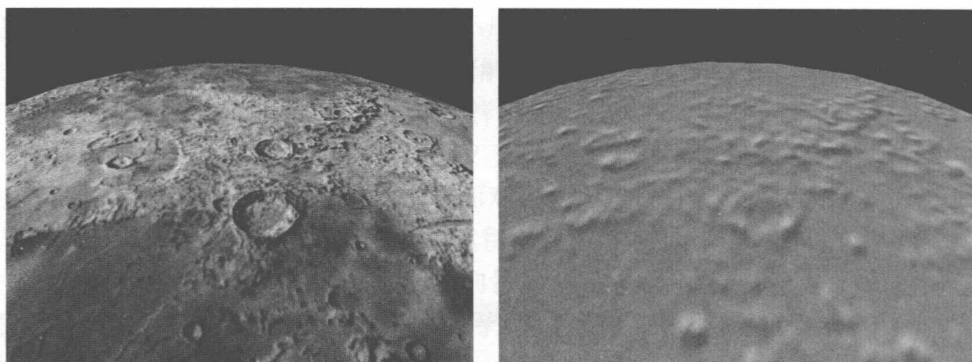


图 10.6 使用月面纹理的球体（左）和使用法线贴图的球体（右）

当我们在球体上使用法线贴图（而不是纹理）时，我们会得到图 10.7 所示的结果。尽管它不像标准纹理那么真实（现在），但是现在它确实响应了光照变化。图 10.7 的第一张图像中从左侧进行光照，第二张图像中则从右侧进行光照。请注意蓝色和黄色箭头所示部分展示了山脊周围漫反射光的变化以及镜面反射高光的移动。

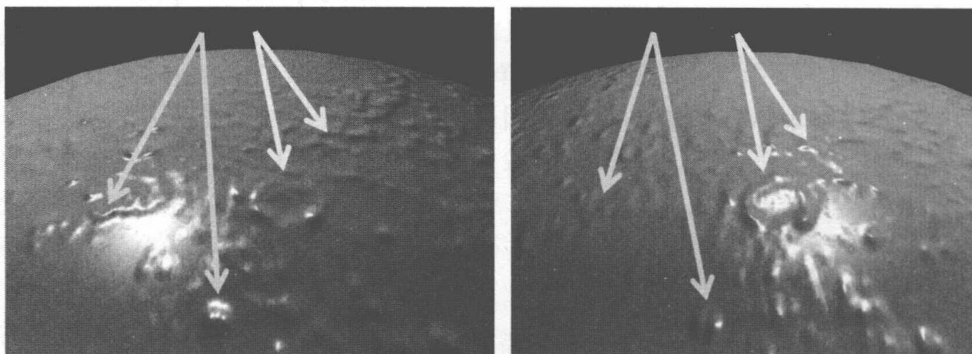


图 10.7 法线贴图对月球的影响

图 10.8 展示了在使用 Phong 光照模型的情况下，将法线贴图与标准纹理相结合的效果。月球的图像通过漫射区域进行了增强，镜面高光区域也会响应光源的移动（或相机或物体移动）。两个图像中的光照分别来自左侧和右侧。

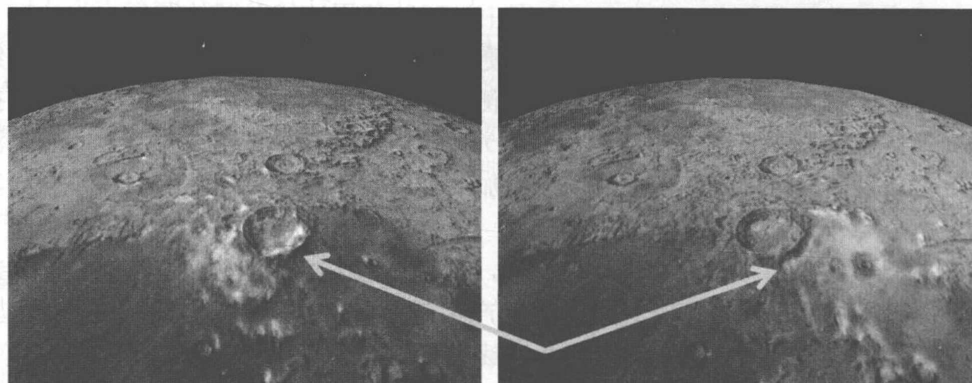


图 10.8 纹理加法线贴图，分别从左侧和右侧进行光照

我们的程序现在需要两个纹理——一个用于月球表面图像，一个用于法线贴图——因此需要有两个采样器。片段着色器使用之前在 7.6 节中所描述的技术，将纹理颜色与经光照计算所得的颜色进行混合，如程序 10.3 所示。

程序 10.3 纹理加法线贴图

```
// 片段着色器中的变量和结构与之前相同，加上
layout (binding=0) uniform sampler2D s0;      // 法线贴图
layout (binding=1) uniform sampler2D s1;      // 纹理
void main(void)
{ // 计算与之前相同，直到

  vec3 N = calcNewNormal();
  vec4 texel = texture(s1,tc);          // 标准纹理
  . . .
  // 反射计算与之前相同，然后混合结果
  fragColor = globalAmbient +
     texel * (light.ambient + light.diffuse * max(cosTheta,0.0)
     + light.specular * pow(max(cosPhi,0.0), material.shininess));
}
```

有趣的是，法线贴图可以从多级渐远纹理贴图（Mipmapping）中受益，因为在第 5 章中看到的纹理化产生的“锯齿”伪影，在使用纹理图像进行法线贴图时也会发生。图 10.9 分别展示了未使用多级渐远纹理贴图和使用了多级渐远纹理贴图进行法线贴图的月球。尽管在静止的图像中不容易观察到，但是左边的球体（未使用多级渐远纹理贴图）周边有闪烁的伪影。

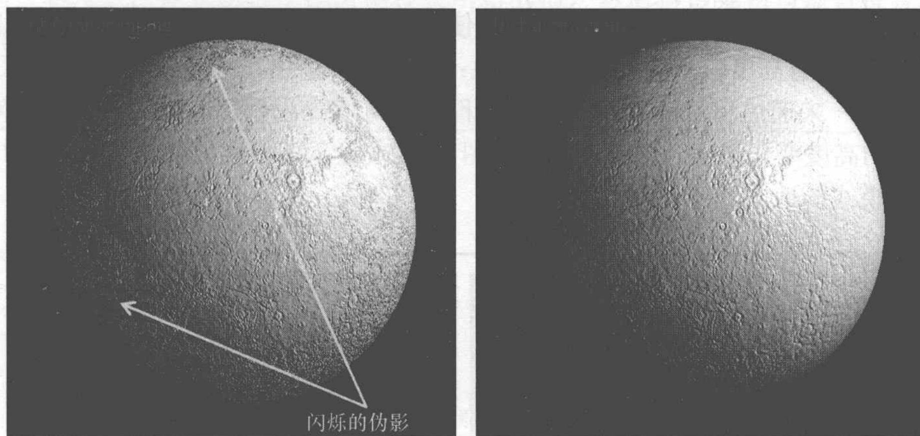


图 10.9 发现贴图伪影，以及使用多级渐远纹理贴图校正后的图像

对于法线贴图而言，各向异性过滤（AF）更有效，它不但减少了闪烁的伪影，同时还保留了细节，如图 10.10 所示（比较右下角边缘的细节）。图 10.11 中展示了使用相等的纹理权重和光照权重，光照应用了法线贴图及 AF 的情况下得到的结果。

最终的渲染结果并不完美。无论光照如何，原始纹理图像中出现的阴影仍将显示在渲染结果上。此外，虽然法线贴图可以影响漫反射和镜面反射效果，但它无法投射阴影。因此，当表面特征较小时，最适用法线贴图。

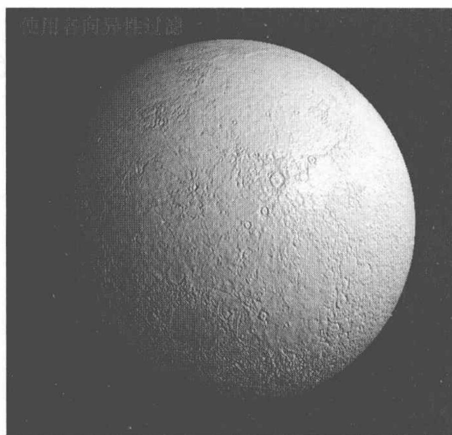


图 10.10 使用各向异性过滤进行法线贴图

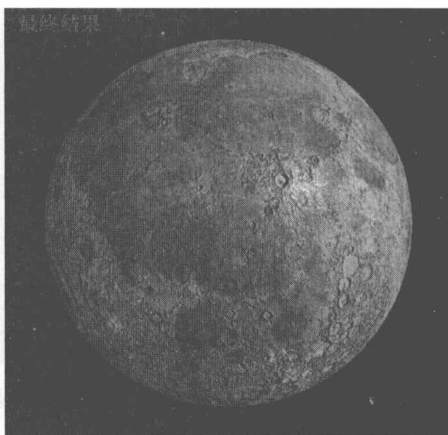


图 10.11 纹理加各向异性过滤法线贴图

10.3 高度贴图

现在我们扩展法线贴图的概念——从纹理图像用于扰动法向量到扰乱顶点位置本身。实际上，以这种方式修改对象的几何体具有一定的优势，例如使表面特征沿着对象的边缘可见，并使特征能够响应阴影贴图。我们将会看到，它还可以帮助构建地形。

一种实用的方法是使用纹理图像来存储高度值，然后使用该高度值来提升（或降低）顶点位置。含有高度信息的图像称为**高度图**，使用高度图更改对象的顶点的方法称为**高度贴图**[①]。高度图通常将高度信息编码为灰度颜色：(0,0,0)（黑色）=低高度，(1,1,1)（白色）=高高度。这样一来通过算法或使用“画图”程序就可以轻松创建高度图。图像的对比度越高，其表示的高度变化越大。这些概念将在图 10.12（显示随机生成的地图）和图 10.13（显示有组织的模式的地图）中说明。

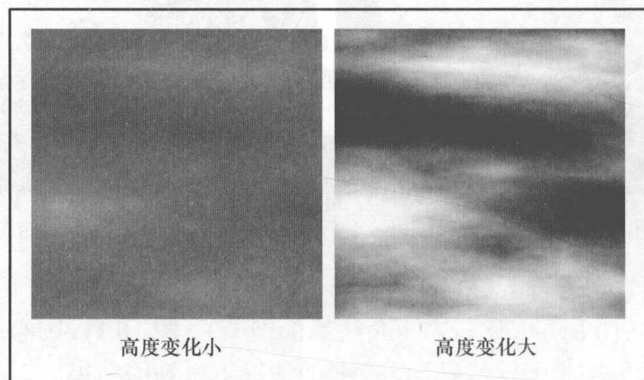


图 10.12 高度图示例

① 这里使用了高度贴图说法，通过纹理图像更改顶点的方法一般称为位移贴图/置换贴图。高度图除了用于位移贴图/置换贴图，有时也用于视差贴图，请读者阅读时注意区别。——译者注

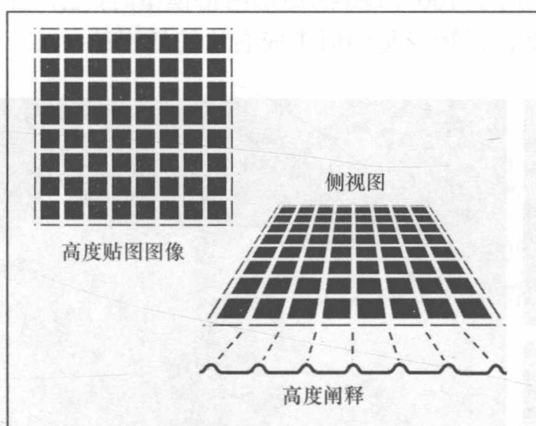


图 10.13 高度贴图阐释

改变顶点位置是否有用取决于改变的模型。顶点操作可以在顶点着色器中轻松完成，当模型顶点细节级别够高（例如在足够高精度的球体中）时，改变顶点高度的方法效果很好。但是，当模型的顶点数量很少（例如立方体的角）时，渲染对象的表面需要依赖于光栅器中的顶点插值来填充细节。当顶点着色器中可用于改变高度的顶点很少时，许多像素的高度将无法从高度图中检索，而需要由插值生成，从而导致表面细节较差。当然，在片段着色器中是不可能进行顶点操作的，因为这时顶点已被光栅化为像素位置。

程序 10.4 展示了一个将顶点“向外”（即在表面法向量的方向上）移动的顶点着色器代码。它通过将顶点法向量乘以从高度图检索所得的值，然后将该乘积与顶点位置相加，以“向外”移动顶点。

程序 10.4 顶点着色器中的高度贴图

```
#version 430

layout (location=0) in vec3 vertPos;
layout (location=1) in vec2 texCoord;
layout (location=2) in vec3 vertNormal;

out vec2 tc;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

layout (binding=0) uniform sampler2D t;     // 用于纹理
layout (binding=1) uniform sampler2D h;     // 用于高度图

void main(void)
{ // "p"是高度图所改变的顶点位置
  // 由于高度图是灰度图，因此使用其任何颜色分量
  // 都可以（我们使用"r"）。除以 5.0 用来调整高度
  vec4 p = vec4(vertPos,1.0) + vec4( (vertNormal * ((texture(h, texCoord).r) / 5.0f)),1.0f );
  tc = tex_coord;
  gl_Position = proj_matrix * mv_matrix * p;
}
```

图 10.14（见彩插）展示了通过在画图程序中涂鸦创建的简单高度图（左上角）。高度图图像中还绘制了一个白色矩形。绿色版本的高度图（左下角）用作纹理。使用程序 10.4 中

展示的着色器将高度图应用于 100×100 的矩形网格模型时，会产生类似“地形”的感觉（如图 10.14 右图所示）。注意白色矩形是如何生成右边的悬崖的。

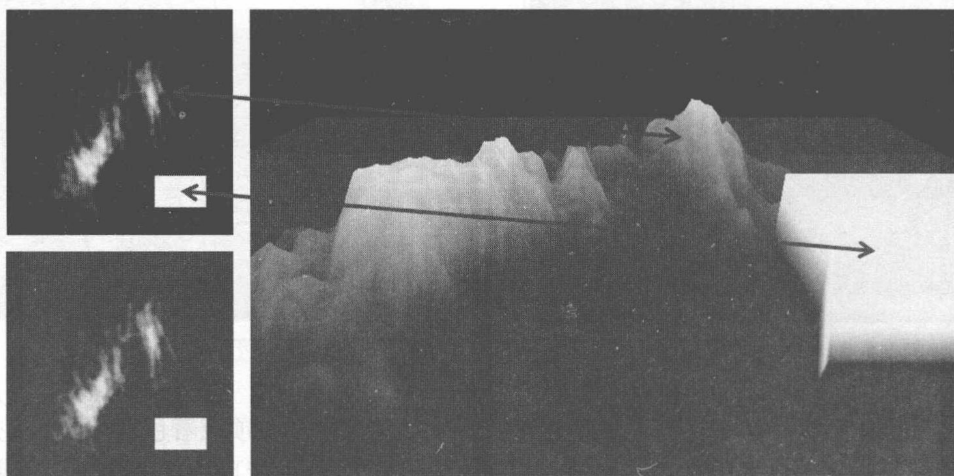


图 10.14 地形，在顶点着色器中进行高度贴图

图 10.14 展示的渲染结果还算可以，因为模型（网格和球体）有足够数量的顶点来对高度贴图值进行采样。也就是说，模型具有大量的顶点，而高度图相对粗糙并且以低分辨率充分地采样。然而，仔细观察仍然会发现存在分辨率伪影，例如沿图 10.14 中地形右侧凸起的矩形盒子的左下边缘。凸起的矩形盒子两侧看起来不是完美矩形，而且颜色有渐变效果，其原因是底层网格 100 像素×100 像素的分辨率无法与高度图中的白色矩形完全对齐，从而导致纹理的光栅化坐标沿侧面产生伪影。

当尝试将其应用于要求更严苛的高度贴图时，在顶点着色器中进行高度贴图的限制会进一步暴露。考虑图 10.5 中展示的月球图像。法线贴图在捕获图像细节方面表现非常出色（如图 10.9 和图 10.11 所示），而且由于它是灰度图，因此尝试将其作为高度图应用似乎很自然。但是，基于顶点着色器的高度贴图会无法胜任这个任务，因为顶点着色器中采样的顶点数（即使对于精度=500 的球体）比起图像中的细节级别，仍然太少。相较之下，法线贴图能够很好地捕获细节，因为在片段着色器中对法线贴图的采样是像素级的。

我们将会在之后的第 12 章继续学习高度图，在那里我们会了解使用曲面细分着色器生成大量顶点的方法。

补充说明

凹凸贴图或法线贴图的一个基本限制是，虽然它们能够在渲染对象的内部提供表面细节的外观，但是物体轮廓（外边界）无法显示这些细节（它保持平滑）。高度贴图在用于实际修改顶点位置时修复了这个缺陷，但它也有其自身的局限性。正如我们将在本书后面看到的，有时可以使用几何着色器或曲面细分着色器来增加顶点的数量，使高度贴图更加实用、有效。

我们冒昧地简化了一些凹凸贴图和法线贴图计算。在重要应用中可以使用更准确和/或更有效的解决方案[BN12]。

习题

10.1 使用程序 10.1 进行试验，修改片段着色器中的设置和/或计算并观察结果。

10.2 使用绘图程序生成一份高度图并在程序 10.4 中使用它。尝试识别由于顶点着色器无法充分采样高度图而缺少细节的位置。你可能会发现使用高度图图像文件在对地形进行纹理化时也很有用，如图 10.14 所示（或者使用可以暴露表面结构的图案，例如网格），这样你就可以看到所生成地形中的山和谷。

10.3 （项目）向程序 10.4 中添加光照，以便进一步暴露高度贴图地形的表面结构。

10.4 （项目）为练习 10.3 中的代码添加阴影贴图，从而使高度贴图地形投射阴影。

参考资料

[BL78] J. Blinn, “Simulation of Wrinkled Surfaces,” *Computer Graphics* 12, no. 3 (1978): 286–292.

[BN12] E. Bruneton and F. Neyret, “A Survey of Non-Linear Pre-Filtering Methods for Efficient and Accurate Surface Shading,” *IEEE Transactions on Visualization and Computer Graphics* 18, no. 2 (2012).

[GI16] GNU Image Manipulation Program, accessed October 2018.

[GP16] GIMP Plugin Registry, normalmap plugin, accessed October 2018.

[HT16] J. Hastings-Trew, JHT’s Planetary Pixel Emporium, accessed October 2018.

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[ME11] E. Meiri, OGLdev tutorial 26, 2011, accessed October 2018.

[PH16] Adobe Photoshop, accessed October 2018.

[SS16] SS Bump Generator, accessed October 2018.

第 11 章　参数曲面

在 20 世纪 50 年代和 60 年代在雷诺公司工作期间，皮埃尔·贝塞尔（Pierre Bézier）开发了用于设计汽车车身的软件系统。他的程序利用了 Paul de Casteljau 之前开发的数学方程组，后者曾为竞争对手雪铁龙汽车制造商[BE72, DC63]工作。de Casteljau 方程仅使用几个标量参数描述曲线，同时使用一种高明的的递归算法，称为“de Casteljau 算法”，就可以生成任意精度的曲线。现在它们分别被称为“贝塞尔曲线”和“贝塞尔曲面”，这些方法通常用于高效地对各种曲面 3D 物体进行建模。

11.1　二次贝塞尔曲线

二次贝塞尔曲线由一组参数方程定义，方程组中使用 3 个**控制点**指定特定的曲线的形状，每个控制点都是 2D 空间中的一个点。①考虑图 11.1 中所示的一组 3 个点[p_0，p_1，p_2]。

通过引入参数 t，我们可以构建一个用来定义曲线的参数方程组。t 表示从一个控制点到另一控制点间线段距离的分数。对于在线段上的点，t 的值在[0…1]的范围内。图 11.2 显示了一个这样的值：$t = 0.75$，分别应用于连接 p_0-p_1 和 p_1-p_2 的线段。通过 t 在两条原始线段上定义了两个新点 $p_{01}(t)$和 $p_{12}(t)$。我们对连接两个新点 $p_{01}(t)$和 $p_{12}(t)$的线段重复该过程，产生点 $P(t)$，其中沿线段 $p_{01}(t)$和 $p_{12}(t)$在 $t = 0.75$ 得到点 $P(t)$。$P(t)$是最终得到的曲线上的点，因此用大写字母 P 表示。

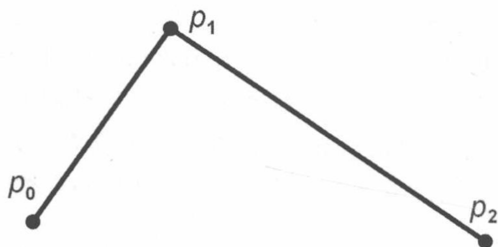


图 11.1　贝塞尔曲线的控制点

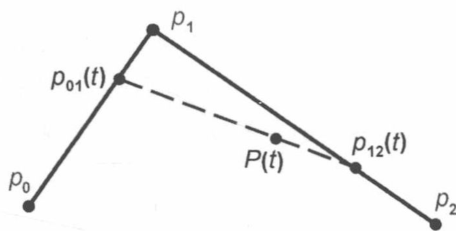


图 11.2　参数位置处的点 $t = 0.75$

针对各种 t 值收集大量的点 $P(t)$，则会产生一条曲线，如图 11.3 所示。采样的 t 的参数值越多，生成的点 $P(t)$越多，得到的曲线则越平滑。

现在可以导出二次贝塞尔曲线的分析定义。首先，我们注意到连接两个点 p_a 和 p_b 的线段 p_a–p_b 上的任意点 p 可以用参数 t 表示如下：

$$p(t) = tp_a + (1-t)p_b$$

① 当然，曲线可以存在于 3D 空间中。然而，二次曲线完全位于 2D 平面内。

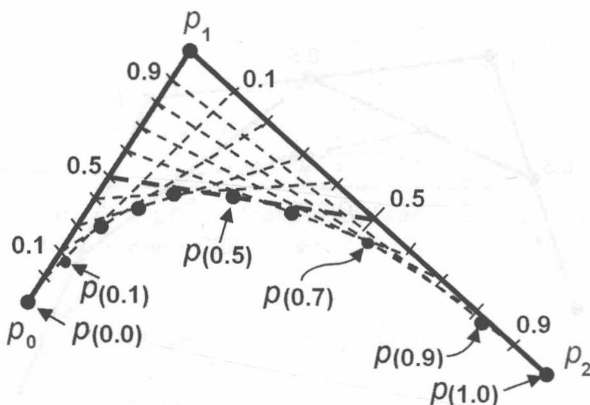


图 11.3 建立二次贝塞尔曲线

使用该等式，我们解出点 p_{01} 和 p_{12}（分别在 p_0–p_1 和 p_1–p_2 上的点）如下：

$$p_{01}(t) = tp_1 + (1-t)p_0$$
$$p_{12}(t) = tp_2 + (1-t)p_1$$

同理，在这两点所连接的线段上的点可以表示为：

$$P(t) = tp_{12}(t) + (1-t)p_{01}(t)$$

替换 p_{12} 和 p_{01} 的定义得：

$$P(t) = t[tp_2 + (1-t)p_1] + (1-t)[tp_1 + (1-t)p_0]$$

分解并重新合并各项可得：

$$P(t) = (1-t)^2 p_0 + (-2t^2 + 2t)p_1 + t^2 p_2$$

或

$$P(t) = \sum_{i=0}^{2} p_i B_i(t)$$

其中

$$B_0(t) = (1-t)^2$$
$$B_1(t) = -2t^2 + 2t$$
$$B_2(t) = t^2$$

因此，我们通过控制点的加权和解出曲线上的任意点。加权函数 B 通常被称为“混合函数”（尽管名称“B”实际上源自 Sergei Bernstein [BE16]，他首先描述了这个多项式族）。请注意，混合函数的形式都是二次的，这就是为什么得到的曲线称为二次贝塞尔曲线。

11.2 三次贝塞尔曲线

我们现在将曲线模型扩展到 4 个控制点，就会得到一个三次贝塞尔曲线，如图 11.4 所示。与二次曲线相比，三次贝塞尔曲线能够定义的形状更加丰富，而二次曲线仅限于定义凹形。

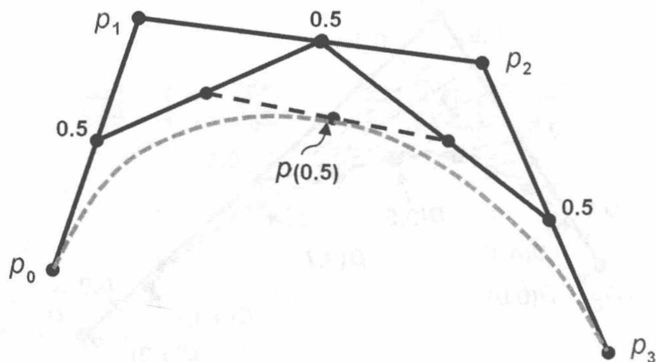


图 11.4 建立一个三次贝塞尔曲线

同二次曲线时的情形，我们可以推导出三次贝塞尔曲线的解析定义：

$$p_{01}(t)=tp_1+(1-t)p_0$$
$$p_{12}(t)=tp_2+(1-t)p_1$$
$$p_{23}(t)=tp_3+(1-t)p_2$$
$$p_{01\text{-}12}(t)=tp_{12}(t)+(1-t)p_{01}(t)$$
$$p_{12-23}(t)=tp_{23}(t)+(1-t)p_{12}(t)$$

曲线上的点则是：

$$P(t)=tp_{12-23}(t)+(1-t)p_{01-12}(t)$$

使用 p_{12-23} 和 p_{01-12} 的定义替换等式中的项，再合并得：

$$P(t)=\sum_{i=0}^{3}p_iB_i(t)$$

其中：

$$B_0(t)=(1-t)^3$$
$$B_1(t)=3t^3-6t^2+3t$$
$$B_2(t)=-3t^3+3t^2$$
$$B_3(t)=t^3$$

渲染贝塞尔曲线时，可以使用许多不同的技术。其中一种方法是，使用固定的增量，在 0.0～1.0 范围内，迭代增加得出 t 的后继值。例如，当增量为 0.1 时，我们可以使用 t 值为 0.0、0.1、0.2、0.3 等的循环。对于 t 的每个值，计算贝塞尔曲线上的对应点，并绘制连接连续点的一系列线段，如图 11.5 中的算法所述。

另一种方法是使用 de Casteljau 算法递归地将曲线对半细分，其中，在每个递归步骤 $t=1/2$。图 11.6 展示了左侧曲线细分后的新三次控制点（q_0，q_1，q_2，q_3），以绿色显示（见彩插）。该算法由 de Casteljau 提出（完整推导见[AS14]）。

算法见图 11.7。该算法重复将曲线段细分为两半的过程，直到每个曲线段足够直，进一步的细分不会产生实际的好处。在极限情况下（随着生成的控制点越来越靠近），曲线段本身实际上与第一个控制点和最后一个控制点（q_0 和 q_3）之间的线段相同。因此，可以通过比较从第一控制点到最后一个控制点的距离与连接 4 个控制点的 3 条线段的长度之和来确

定曲线段是否“足够直”：

$$D_1 = |p_0 - p_1| + |p_1 - p_2| + |p_2 - p_3|$$
$$D_2 = |p_0 - p_3|$$

```
void drawBezierCurve (controlPointVector C)
{   currentPoint = C[0];  // 曲线从第一个控制点开始
    t = 0.0;
    while (t <= 1.0)
    {   //计算混合函数在t时对控制点的加权和，
        //作为曲线的下一个点
        nextPoint = (0,0) ;
        for (int i=0; i<=3; i++)
            nextPoint = nextPoint + (blending(i,t) * C[i]);
        drawLine (currentPoint,nextPoint);
        currentPoint = nextPoint;
        t = t + increment;
}   }

double blending(int i, double t)
{   switch (i)
    {   case 0: return ((1-t)*(1-t)*(1-t));       // (1-t)3
        case 1: return (3*t*(1-t)*(1-t));         // 3t(1-t)2
        case 2: return (3*t*t*(1-t));             // 3t2(1-t)
        case 3: return (t*t*t);                   // t3
}   }
```

图 11.5　渲染贝塞尔曲线的迭代算法

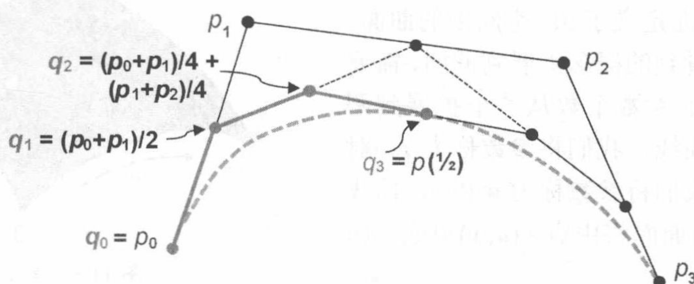


图 11.6　细分三次贝塞尔曲线

当 $D_1 - D_2$ 小于一个足够小的阈值时，进一步的细分就没有意义了。

de Casteljau 算法有一个有趣的特性，它可以在不使用之前描述的混合函数的情况下，生成曲线上所有的点。同时请注意，$p(1/2)$处的中心点是“共享”的，即它既是左细分中最右的控制点，也是右细分中最左的控制点。它可以使用 $t = 1/2$ 处的混合函数或使用由 de Casteljau 导出的公式$(q_2 + r_1)/2$ 来计算。

另请注意，图 11.7 中所示的 subdivide()函数假定传入的参数 p、q 和 r 是“引用”参数，因此，图 11.7 上方列出的 drawBezierCurve 函数对于 subdivide()的调用，导致 subdivide()函数中的计算修改了调用中所传的实际参数。

```
drawBezierCurve(ControlPointVector C)
{   if (C is "straight enough")
        draw line from first to last control point
    else
    {   subdivide(C, LeftC, RightC)
        drawBezierCurve(LeftC)
        drawBezierCurve(RightC)
}   }
subdivide(ControlPointVector p, q, r)
{   // 计算左细分的控制点
    q(0) = p(0)
    q(1) = (p(0)+p(1)) / 2
    q(2) = (p(0)+p(1)) / 4 + (p(1)+p(2)) / 4
    // 计算右细分的控制点
    r(1) = (p(1)+p(2)) / 4 + (p(2)+p(3)) / 4
    r(2) = (p(2)+p(3)) / 2
    r(3) = p(3)
    // 当 t=0.5 时，计算"共享"控制点
    q(3) = r(0) = (q(2)+r(1)) / 2
}
```

图 11.7 贝塞尔曲线的递归细分算法

11.3 二次贝塞尔曲面

贝塞尔曲线定义了曲线(在 2D 或 3D 空间中)，而贝塞尔曲面定义了 3D 空间中的曲面。将我们在曲线中看到的概念扩展到曲面，需要将参数方程组中的参数个数从一个扩展到两个。对于贝塞尔曲线，我们将参数称为 t。对于贝塞尔曲面，我们将参数称为 u 和 v。曲线由点 $P(t)$组成，而曲面将由点 $P(u, v)$组成，如图 11.8 所示。

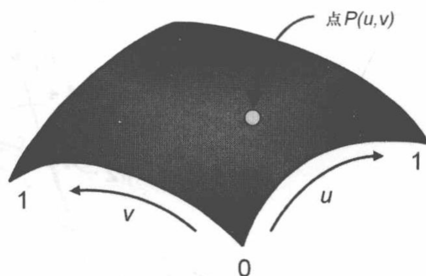


图 11.8 参数曲面

对于二次贝塞尔曲面，每个轴 u 和 v 上有 3 个控制点，总共 9 个控制点。图 11.9（见彩插）使用蓝色展示了一组共 9 个控制点（通常称为控制点"网格"）的示例，以及相应的曲面（红色）。

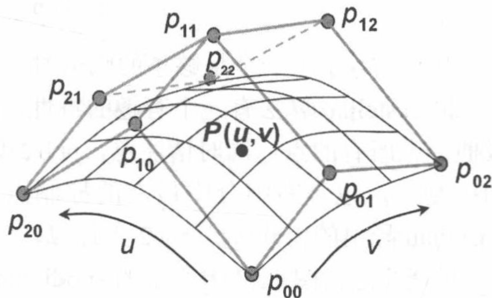


图 11.9 二次贝塞尔控制网格和相应的表面

网格中的 9 个控制点标记为 p_{ij}，其中 i 和 j 分别代表 u 和 v 方向上的索引。每组 3 个相邻控制点（例如（p_{00}，p_{01}，p_{02}））会定义一条贝塞尔曲线。然后将表面上的点 $P(u, v)$定义为两个混合函数的和，一个在 u 方向，一个在 v 方向。则用于构建贝塞尔曲面的两个混合函数的

形式遵循先前为贝塞尔曲线给出的方法：

$$B_0(u) = (1-u)^2$$
$$B_1(u) = -2u^2 + 2u$$
$$B_2(u) = u^2$$
$$B_0(v) = (1-v)^2$$
$$B_1(v) = -2v^2 + 2v$$
$$B_2(v) = v^2$$

接下来生成构成贝塞尔曲面的点 $P(u, v)$。对于每个控制点 p_{ij}，将其与第 i 个混合函数在 u 处的值相乘，再与第 j 个混合函数在 v 处的值相乘。最后将所有控制点的结果求和，生成贝塞尔曲面上的点 $P(u, v)$：

$$P(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} p_{ij} * B_i(u) * B_j(v)$$

组成贝塞尔曲面的生成点集有时会称为补丁。术语“补丁”有时会让人感到困惑，我们稍后在研究曲面细分着色器时会看到（对于实际实现贝塞尔曲面非常有用）。因为通常控制点组成的网格才称为“补丁”。

11.4 三次贝塞尔曲面

从二次曲面到三次曲面需要使用更大的网格——4×4 而非 3×3。图 11.10（见彩插）显示了 16 控制点网格（蓝色）和相应曲面（红色）的示例。

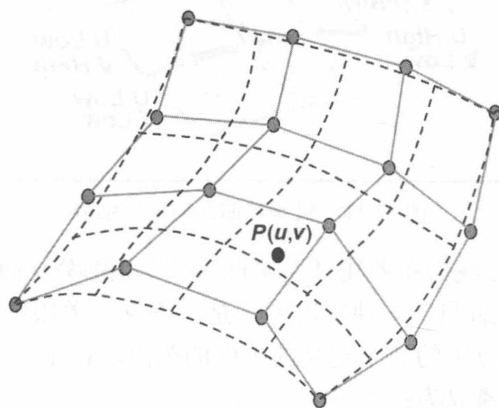


图 11.10 三次贝塞尔控制网格和相应的曲面

同上，我们可以通过组合三次贝塞尔曲线的相关混合函数来推导表面上的点 $P(u, v)$的公式：

$$P(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} p_{ij} * B_i(u) * B_j(v)$$

其中：

$$B_0(u) = (1-u)^3 \qquad B_0(v) = (1-v)^3$$
$$B_1(u) = -3u^3 - 6u^2 + 3u \qquad B_1(v) = -3v^3 - 6v^2 + 3v$$
$$B_2(u) = -3u^3 + 3u^2 \qquad B_2(v) = -3v^3 + 3v^2$$
$$B_3(u) = u^3 \qquad B_3(v) = v^3$$

渲染贝塞尔曲面也可以通过递归细分[AS14]完成，方法是交替地将曲面沿每个维度分成两半，如图 11.11 所示。每个细分产生 4 个新的控制点网格，每个网格包含 16 个点，这些点定义了曲面的一个象限。

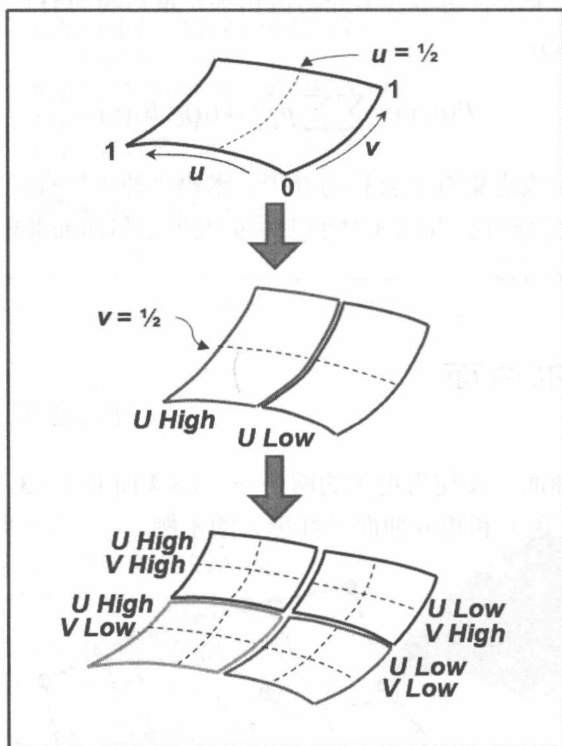


图 11.11 贝塞尔曲面的递归细分

当渲染贝塞尔曲线时，我们在曲线“足够直”时停止细分。而对于贝塞尔曲面，我们在曲面“足够平坦”时停止递归。一种实现方法是，确保子象限控制网格上所有递归生成的点，距由该网格的 4 个角点中的 3 个定义的平面的距离，都小于一个允许的范围。点(x,y,z)与平面(A,B,C,D)之间的距离 d 为：

$$d = \text{abs}\left(\frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}}\right)$$

如果 d 小于某个足够小的阈值，则我们停止细分过程，并简单地使用子象限网格的 4 个角的控制点来绘制两个三角形。

对于贝塞尔曲线，OpenGL 管线的细分阶段为基于图 11.5 中的迭代算法渲染贝塞尔曲面提供了一种有吸引力的替代方法。其策略是让曲面细分生成一个大的顶点网格，然后使用

混合函数将这些顶点重新定位到贝塞尔曲面上，由三次贝塞尔控制点指定。我们在第 12 章中实现了这一点。

补充说明

本章重点介绍参数贝塞尔曲线和曲面的数学基础。我们推迟了在 OpenGL 中呈现其中任何一个的实现，因为实现它们需要适当的曲面细分着色器知识作为载体，我们将在下一章中进行介绍。我们还跳过了一些推导过程，例如递归细分算法。

在 3D 图形中，使用贝塞尔曲线建模对象有许多优点。首先，理论上，这些物体可以任意缩放，并且仍然保持光滑的表面而不“像素化”。其次，许多由复杂曲线组成的物体可以使用贝塞尔控制点集合进行更有效的存储，而不是存储数千个顶点。

除计算机图形和汽车外，贝塞尔曲线还有许多实际应用。在桥梁设计中也可以找到它们的身影，例如耶路撒冷的 Chords Bridge [CB16]。类似的技术也用于构建 TrueType 字体，因此可以将其缩放到任意大小，或者将视角任意拉近观看，而字体边缘始终保持平滑。

习题

11.1 二次贝塞尔曲线仅限于定义完全“凹”或“凸”的曲线。描述（或绘制）一个曲线作为例子，该曲线既不以完全凹的形式，也不以完全凸的方式弯曲，因此无法通过二次贝塞尔曲线进行近似描述。

11.2 使用钢笔或铅笔在一张纸上绘制一组任意 4 个点，按任意顺序编号为 1～4，然后尝试大致绘制一条由这 4 个有序控制点定义的三次贝塞尔曲线。接着重新排列控制点的编号（改变它们的顺序，但不改变它们的位置）并重新绘制新产生的三次贝塞尔曲线。互联网上有许多在线工具可以用于绘制贝塞尔曲线，你可以使用它们来检验你绘制的曲线。

参考资料

[AS14] E. Angel and D. Shreiner, *Interactive Computer Graphics: A Top-Down Approach with WebGL*, 7th ed. (Pearson, 2014).

[BE16] S. Bernstein, Wikipedia, accessed October 2018.

[BE72] P.Bézier, *Numerical Control:Mathematics and Applications* (JohnWiley& Sons, 1972).

[CB16] Chords Bridge, Wikipedia, accessed October 2018.

[DC63] P. de Casteljau, “Courbes et surfaces à pôles ,” technical report (A. Citroën, 1963).

第 12 章　曲面细分

术语 Tessellation（镶嵌）是指一大类设计活动，通常是指在平坦的表面上，用各种几何形状的瓷砖相邻排列以形成图案。它的目的可以是艺术性的或实用性的，很多例子可以追溯到几千年前[TS16]。

在 3D 图形学中，Tessellation 指的是有点不同的东西（曲面细分），但显然是由它的经典对应物（镶嵌）启发而成的。在这里，曲面细分指的是生成并且操控大量三角形以渲染复杂的形状和表面，尤其是使用硬件进行渲染。曲面细分是 OpenGL 核心近期才增加的新功能，在 2010 年的 4.0 版本中出现。①

12.1　OpenGL 中的曲面细分

OpenGL 对硬件曲面细分的支持，通过 3 个管线阶段提供：

（1）曲面细分控制着色器；

（2）曲面细分器；

（3）曲面细分评估着色器。

第（1）和第（3）阶段是可编程的；而中间的第（2）阶段不是。为了使用曲面细分，程序员通常会提供控制着色器和评估着色器。

曲面细分器（其全名是*曲面细分图元生成器*，或 TPG）是硬件支持的引擎，可以生成固定的三角形网格。②控制着色器允许我们配置曲面细分器要构建什么样的三角形网格。然后，评估着色器允许我们以各种方式操控网格。然后，被操控过的三角形网格，会作为通过管线前进的顶点的源数据。回想一下图 2.2，在管线上，曲面细分着色器位于顶点着色器和几何着色器阶段之间。

让我们从一个简单的应用程序开始，该应用程序只使用曲面细分器创建顶点的三角形网格，然后在不进行任何操作的情况下显示它。为此，我们需要以下模块。

（1）C++/OpenGL 应用程序：

创建一个摄像机和相关的 MVP 矩阵，视图（v）和投影（p）矩阵确定摄像机朝向，模型（m）矩阵可用于修改网格的位置和方向。

（2）顶点着色器：

在这个例子中基本上什么都不做，顶点将在曲面细分器中生成。

（3）曲面细分控制着色器：

---

① GLU 工具集之前已经包含了一个名为 gluTess 的曲面细分实用程序。2001 年，Radeon 发布了第一款带有曲面细分支持的商用图形卡，但很少有工具可以利用它。

② 或线段，但我们将专注于三角形。

指定曲面细分器要构建的网格。

（4）曲面细分评估着色器：

将 MVP 矩阵应用于网格中的顶点。

（5）片段着色器：

只需为每个像素输出固定颜色。

程序 12.1 显示了整个应用程序的代码。即使像这样的简单示例也相当复杂，因此许多代码元素都需要解释。请注意，这是我们第一次使用除顶点和片段着色器之外的组件构建 GLSL 渲染程序。因此，我们实现了 createShaderProgram()的 4 参数重载版本。

程序 12.1 基本曲面细分器网格

```
C++ / OpenGL 应用程序

GLuint createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *fp) {
  string vertShaderStr = readShaderSource(vp);
  string tcShaderStr = readShaderSource(tCS);
  string teShaderStr = readShaderSource(tES);
  string fragShaderStr = readShaderSource(fp);

  const char *vertShaderSrc = vertShaderStr.c_str();
  const char *tcShaderSrc = tcShaderStr.c_str();
  const char *teShaderSrc = teShaderStr.c_str();
  const char *fragShaderSrc = fragShaderStr.c_str();

  GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
  GLuint tcShader = glCreateShader(GL_TESS_CONTROL_SHADER);
  GLuint teShader = glCreateShader(GL_TESS_EVALUATION_SHADER);
  GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);

  glShaderSource(vShader, 1, &vertShaderSrc, NULL);
  glShaderSource(tcShader, 1, &tcShaderSrc, NULL);
  glShaderSource(teShader, 1, &teShaderSrc, NULL);
  glShaderSource(fShader, 1, &fragShaderSrc, NULL);

  glCompileShader(vShader);
  glCompileShader(tcShader);
  glCompileShader(teShader);
  glCompileShader(fShader);

  GLuint vtfprogram = glCreateProgram();
  glAttachShader(vtfprogram, vShader);
  glAttachShader(vtfprogram, tcShader);
  glAttachShader(vtfprogram, teShader);
  glAttachShader(vtfprogram, fShader);
  glLinkProgram(vtfprogram);
  return vtfprogram;
}

void init(GLFWwindow* window) {
  . . .
  renderingProgram = createShaderProgram("vertShader.glsl",
    "tessCShader.glsl", "tessEShader.glsl", "fragShader.glsl");
}

void display(GLFWwindow* window, double currentTime) {
  . . .
```

```
  glUseProgram(renderingProgram);
  . . .
  glPatchParameteri(GL_PATCH_VERTICES, 1);
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
  glDrawArrays(GL_PATCHES, 0, 1);
}
```

顶点着色器

```
#version 430
uniform mat4 mvp_matrix;
void main(void) { }
```

曲面细分控制着色器

```
#version 430
uniform mat4 mvp_matrix;
layout (vertices = 1) out;

void main(void)
{ gl_TessLevelOuter[0] = 6;
  gl_TessLevelOuter[1] = 6;
  gl_TessLevelOuter[2] = 6;
  gl_TessLevelOuter[3] = 6;
  gl_TessLevelInner[0] = 12;
  gl_TessLevelInner[1] = 12;
}
```

曲面细分评估着色器

```
#version 430
uniform mat4 mvp_matrix;
layout (quads, equal_spacing, ccw) in;

void main (void)
{ float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;
  gl_Position = mvp_matrix * vec4(u,0,v,1);
}
```

片段着色器

```
#version 430
out vec4 color;
uniform mat4 mvp_matrix;

void main(void)
{ color = vec4(1.0, 1.0, 0.0, 1.0); // 黄色
}
```

得到的输出网格如图 12.1 所示（见彩插）。

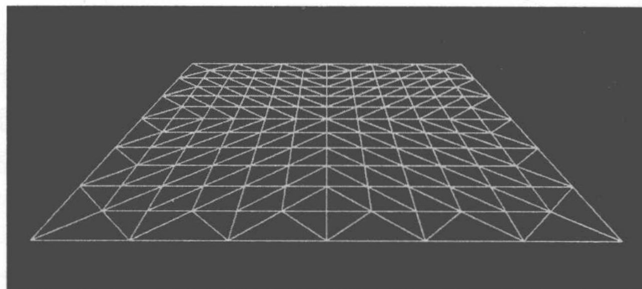


图 12.1 Tessellator 三角形网格输出

曲面细分器生成由两个参数定义的顶点网格：*内层级别*和*外层级别*。在这种情况下，内层级别为 12，外层级别为 6——网格的外边缘被分为 6 段，而跨越内部的线被分为 12 段。

程序 12.1 中的特别相关的新结构被高亮显示。让我们首先讨论第一部分——C++/OpenGL 代码。

编译这两个新着色器，跟顶点和片段着色器完全相同。然后将它们附加到同一个渲染程序，并且链接调用保持不变。唯一的新项目是用于指定要实例化的着色器类型的常量——新常量如下：

```
GL_TESS_CONTROL_SHADER
GL_TESS_EVALUATION_SHADER
```

请注意 display()函数中的新项目。glDrawArrays()调用现在指定 GL_PATCHES。当使用曲面细分时，从 C++/OpenGL 应用程序发送到管线（即在 VBO 中）的顶点不会被渲染，但通常会被当作控制点，就像我们在贝塞尔曲线中看到的那些一样。一组控制点被称作“补丁”，并且在使用曲面细分的代码段中，GL_PATCHES 是唯一允许的图元类型。“补丁”中顶点的数量在 glPatchParameteri()的调用中指定。在这个特定示例中，没有任何控制点被发送，但我们仍然需要指定至少一个。类似地，在 glDrawArrays()调用中，我们指示起始值为 0，顶点数量为 1，即使我们实际上没有从 C++程序发送任何顶点。

对 glPolygonMode()的调用指定了如何光栅化网格。默认值为 GL_FILL。而我们的代码中显示的是 GL_LINE，如我们在图 12.1 中看到的那样，它只会导致连接线被光栅化（因此我们可以看到由曲面细分器生成的网格本身）。如果我们将该行代码更改为 GL_FILL（或将其注释掉，从而使用默认行为 GL_FILL），我们将得到如图 12.2 所示的版本。

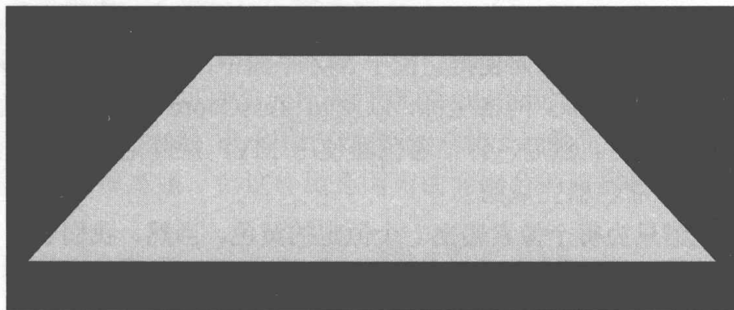


图 12.2　使用 GL_FILL 渲染的细分网格

现在让我们来过一遍 4 个着色器。如前所述，顶点着色器几乎没什么可做的，因为 C++/OpenGL 应用程序没有提供任何顶点。它包含的是一个统一变量声明，以和其他着色器相匹配，以及一个空的 main()。在任何情况下，所有着色器程序都必须包含顶点着色器。

曲面细分控制着色器指定曲面细分器要生成的三角形网格的拓扑结构。通过将值分配给名为 gl_ TessLevelxxx 的保留字，设置 6 个“级别”参数——两个“内部”和 4 个“外部”级别。我们这里细分了一个由三角形组成的大矩形网格，称为四边形。[①]级别参数告诉曲面

① 曲面细分器还能够构建由三角形组成的三角形网格，但本书中未对此进行介绍。

细分器在形成三角形时如何细分网格，它们的排列如图 12.3 所示。

请注意控制着色器中的代码行：

```
layout (vertices=1) out;
```

这与之前的 GL_PATCHES 讨论有关，用来指定从顶点着色器传递给控制着色器（以及“输出”给评估着色器）的每个“补丁”的顶点数。在我们现在这个程序中没有任何顶点，但我们仍然必须指定至少一个，因为它也会影响控制着色器被执行的次数。稍后这个值将反映控制点的数量，并且必须与 C++/OpenGL 应用程序中 glPatchParameteri()调用中的值匹配。

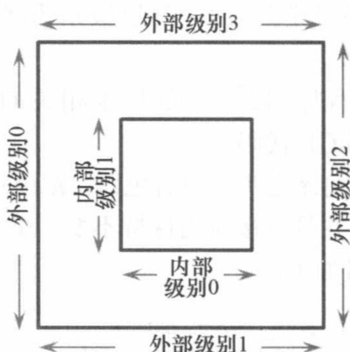


图 12.3 细分级别

接下来让我们看一下曲面细分评估着色器。它以一行代码开头，形如：

```
layout (quads, equal_spacing, ccw) in;
```

乍一看这好像与控件着色器中的“out”布局语句有关，但实际上它们是无关的。相反，这行代码是我们指示曲面细分器去生成排列在一个大矩形（“四边形”）中顶点的位置。它还指定了细分线段（包括内部和外部）具有相等的长度（稍后我们将看到长度不等的细分的应用场景）。“ccw”参数指定生成曲面细分网格顶点的缠绕顺序（在当前情况下，是逆时针）。

然后，由曲面细分器生成的顶点被发送到评估着色器。因此，评估着色器既可以从控制着色器（通常作为控制点），又可以从曲面细分器（曲面细分网格）接收顶点。在程序 12.1 中，仅从曲面细分器接收顶点。

评估着色器对曲面细分器生成的每个顶点执行一次。可以使用内置变量 gl_TessCoord 访问顶点位置。曲面细分网格的朝向使得它位于 *X*-*Z* 平面中，因此 gl_TessCoord 的 *X* 和 *Y* 分量被应用于网格的 *X* 和 *Z* 坐标。网格坐标，以及 gl_TessCoord 的值，范围为 0.0～1.0（这在计算纹理坐标时会很方便）。然后，评估着色器使用 MVP 矩阵定向每个顶点（这在前面章节的示例中，是由顶点着色器完成的）。

最后，片段着色器只为每个像素输出一个恒定的黄色。当然，我们也可以使用它来为我们的场景应用纹理或光照，就像我们在前面的章节中看到的那样。

12.2 贝塞尔曲面细分

现在让我们扩展我们的程序，使它将我们简单的矩形网格转换为贝塞尔曲面。细分网格应该为我们提供了足够的顶点来对曲面进行采样（如果我们想要更多的话，我们可以增加内部/外部细分级别）。我们现在需要的是通过管线发送控制点，然后使用这些控制点执行计算以将细分网格转换为我们所需的贝塞尔曲面。

假设我们希望建立一个立方体贝塞尔曲面，我们将需要 16 个控制点。我们可以通过 VBO 从 C++端发送它们，或者我们可以在顶点着色器中硬编码写死它们。图 12.4 概述了来自 C++

端的控制点的过程。

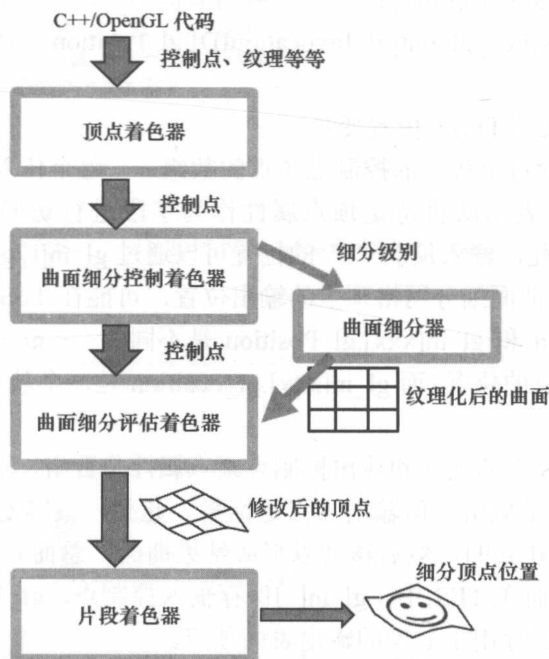


图 12.4 贝塞尔曲面的曲面细分概述

现在是更准确地解释曲面细分控制着色器（TCS）如何工作的好时机。与顶点着色器类似，TCS 对每个传入顶点执行一次。另外，回想一下第 2 章，OpenGL 提供了一个名为 gl_VertexID 的内置变量，它保存一个计数器，指示顶点着色器当前正在执行哪次调用。曲面细分控制着色器中存在一个类似的内置变量 gl_InvocationID。

曲面细分的一个强大功能是 TCS（以及 TES）着色器可以同时访问数组中的所有控制点顶点。首先，当每个调用都可以访问所有顶点时，TCS 对每个顶点执行一次可能会让人感到困惑。在每个 TCS 调用中，冗余地在赋值语句中指定曲面细分级别也是违反直觉的。尽管所有这些看起来都很奇怪，但这样做是因为曲面细分的架构设计使得 TCS 调用可以并行运行。

OpenGL 提供了几个用于 TCS 和 TES 着色器的内置变量。我们已经提到过的是 gl_InvocationID，当然还有 gl_TessLevelInner 和 gl_TessLevelOuter。以下是一些最有用的内置变量的更多细节和描述。

曲面细分控制着色器（TCS）内置变量。

- gl_in[ ]——包含每个传入的控制点顶点的数组——每个传入顶点是一个数组元素。可以使用“.”表示法将特定顶点属性作为字段进行访问。一个内置属性是 gl_Position——因此，输入顶点“i”的位置可以通过 gl_in[i].gl_Position 访问。
- gl_out[ ]——用于将输出控制点的顶点发送到 TES 的一个数组——每个输出顶点是一个数组元素。可以使用“.”表示法将特定顶点属性作为字段进行访问。一个内置属性是 gl_Position——因此，输出顶点“i”的位置可以通过 gl_out[i].gl_Position 访问。

- gl_InvocationID——整型 ID 计数器，指示 TCS 当前正在执行哪个调用。一个常见的用途是用于传递顶点属性；例如，将当前调用的顶点位置从 TCS 传递到 TES 可以用如下方式完成：gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position。

曲面细分评估着色器（TES）内置变量。

- gl_in[ ]——包含每个传入的控制点顶点的数组——每个传入顶点是一个数组元素。可以使用“.”表示法将特定顶点属性作为字段进行访问。一个内置属性是 gl_Position——因此，输入顶点“i”的位置可以通过 gl_in[i].gl_Position 访问。
- gl_Position——曲面细分网格顶点的输出位置，可能在 TES 中被修改。重要的是要注意 gl_Position 和 gl_in[xxx].gl_Position 是不同的——gl_Position 是起源于曲面细分器的输出顶点的位置，而 gl_in[xxx].gl_Position 是一个从 TCS 进入 TES 的控制点顶点位置。

值得注意的是，TCS 中的输入和输出控制点顶点属性是数组。不同的是，TES 中的输入控制点顶点和顶点属性是数组，但输出顶点是标量。此外，很容易混淆哪些顶点来自于控制点，哪些顶点是细分建立的，然后移动以形成结果曲面。总而言之，TCS 的所有顶点输入和输出都是控制点，而在 TES 中，gl_in[ ]保存输入控制点，gl_TessCoord 保存输入的细分网格点，gl_Position 保存用于渲染的输出表面顶点。

我们的曲面细分控制着色器现在有两个任务：指定曲面细分级别并将控制点从顶点着色器传递到评估着色器。然后，评估着色器可以根据贝塞尔控制点修改网格点（gl_TessCoords）的位置。

程序 12.2 显示了所有 4 个着色器——顶点、TCS、TES 和片段——用于指定控制点补丁，生成平坦的曲面细分顶点网格，在控制点指定的曲面上重新定位这些顶点，并使用纹理图像绘制生成的曲面。它还显示了 C++/OpenGL 应用程序的相关部分，特别是在 display()函数中。在此示例中，控制点源自顶点着色器（它们在那里硬编码写死），而不是从 C++/OpenGL 应用程序进入 OpenGL 管线。代码后面会讲述其他详细信息。

程序 12.2 贝塞尔曲面的曲面细分

顶点着色器

```
#version 430
out vec2 texCoord;
uniform mat4 mvp_matrix;
layout (binding = 0) uniform sampler2D tex_color;

void main(void)
{ // 这次由顶点着色器指定和发送控制点
  const vec4 vertices[ ] =
  vec4[ ] (vec4(-1.0, 0.5, -1.0, 1.0), vec4(-0.5, 0.5, -1.0, 1.0),
           vec4( 0.5, 0.5, -1.0, 1.0), vec4( 1.0, 0.5, -1.0, 1.0),

           vec4(-1.0, 0.0, -0.5, 1.0), vec4(-0.5, 0.0, -0.5, 1.0),
           vec4( 0.5, 0.0, -0.5, 1.0), vec4( 1.0, 0.0, -0.5, 1.0),

           vec4(-1.0, 0.0, 0.5, 1.0), vec4(-0.5, 0.0, 0.5, 1.0),
           vec4( 0.5, 0.0, 0.5, 1.0), vec4( 1.0, 0.0, 0.5, 1.0),
```

```
        vec4(-1.0, -0.5, 1.0, 1.0), vec4(-0.5, 0.3, 1.0, 1.0),
        vec4( 0.5, 0.3, 1.0, 1.0), vec4( 1.0, 0.3, 1.0, 1.0) );

  // 为当前顶点计算合适的纹理坐标，从[-1...+1]转换到[0...1]
  texCoord = vec2((vertices[gl_VertexID].x + 1.0) / 2.0, (vertices[gl_VertexID].z + 1.0) / 2.0);
  gl_Position = vertices[gl_VertexID];
}
```

曲面细分控制着色器

```
#version 430
in vec2 texCoord[ ];
out vec2 texCoord_TCSout[ ];  // 以标量形式从顶点着色器传来的纹理坐标输出，以数组形式被接收，然后被发送给评估着色器

uniform mat4 mvp_matrix;
layout (binding = 0) uniform sampler2D tex_color;
layout (vertices = 16) out;      // 每个补丁有16个控制点

void main(void)
{ int TL = 32;                   // 曲面细分级别都被设置为这个值
  if (gl_InvocationID == 0)
  { gl_TessLevelOuter[0] = TL; gl_TessLevelOuter[2] = TL;
    gl_TessLevelOuter[1] = TL; gl_TessLevelOuter[3] = TL;
    gl_TessLevelInner[0] = TL; gl_TessLevelInner[1] = TL;
  }
  // 将纹理和控制点传递给TES
  texCoord_TCSout[gl_InvocationID] = texCoord[gl_InvocationID];
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

曲面细分评估着色器

```
#version 430
layout (quads, equal_spacing,ccw) in;
uniform mat4 mvp_matrix;
layout (binding = 0) uniform sampler2D tex_color;
in vec2 texCoord_TCSout[ ];
out vec2 texCoord_TESout;              // 以标量形式传来的纹理坐标数组被一个个传出

void main (void)
{ vec3 p00 = (gl_in[0].gl_Position).xyz;
  vec3 p10 = (gl_in[1].gl_Position).xyz;
  vec3 p20 = (gl_in[2].gl_Position).xyz;
  vec3 p30 = (gl_in[3].gl_Position).xyz;
  vec3 p01 = (gl_in[4].gl_Position).xyz;
  vec3 p11 = (gl_in[5].gl_Position).xyz;
  vec3 p21 = (gl_in[6].gl_Position).xyz;
  vec3 p31 = (gl_in[7].gl_Position).xyz;
  vec3 p02 = (gl_in[8].gl_Position).xyz;
  vec3 p12 = (gl_in[9].gl_Position).xyz;
  vec3 p22 = (gl_in[10].gl_Position).xyz;
  vec3 p32 = (gl_in[11].gl_Position).xyz;
  vec3 p03 = (gl_in[12].gl_Position).xyz;
  vec3 p13 = (gl_in[13].gl_Position).xyz;
  vec3 p23 = (gl_in[14].gl_Position).xyz;
  vec3 p33 = (gl_in[15].gl_Position).xyz;

  float u = gl_TessCoord.x;
  float v = gl_TessCoord.y;

  // 立方贝塞尔基础函数
```

```
	float bu0 = (1.0-u) * (1.0-u) * (1.0-u);	// (1-u)^3
	float bu1 = 3.0 * u * (1.0-u) * (1.0-u);	// 3u(1-u)^2
	float bu2 = 3.0 * u * u * (1.0-u);	// 3u^2(1-u)
	float bu3 = u * u * u;	// u^3
	float bv0 = (1.0-v) * (1.0-v) * (1.0-v);	// (1-v)^3
	float bv1 = 3.0 * v * (1.0-v) * (1.0-v);	// 3v(1-v)^2
	float bv2 = 3.0 * v * v * (1.0-v);	// 3v^2(1-v)
	float bv3 = v * v * v;	// v^3

	// 输出曲面细分补丁中的顶点位置
	vec3 outputPosition =
		bu0 * ( bv0*p00 + bv1*p01 + bv2*p02 + bv3*p03 )
	  + bu1 * ( bv0*p10 + bv1*p11 + bv2*p12 + bv3*p13 )
	  + bu2 * ( bv0*p20 + bv1*p21 + bv2*p22 + bv3*p23 )
	  + bu3 * ( bv0*p30 + bv1*p31 + bv2*p32 + bv3*p33 );
	gl_Position = mvp_matrix * vec4(outputPosition,1.0f);

	// 输出插值过的纹理坐标
	vec2 tc1 = mix(texCoord_TCSout[0], texCoord_TCSout[3], gl_TessCoord.x);
	vec2 tc2 = mix(texCoord_TCSout[12], texCoord_TCSout[15], gl_TessCoord.x);
	vec2 tc = mix(tc2, tc1, gl_TessCoord.y);
	texCoord_TESout = tc;
}
```

片段着色器

```
#version 430
in vec2 texCoord_TESout;
out vec4 color;
uniform mat4 mvp_matrix;
layout (binding = 0) uniform sampler2D tex_color;

void main(void)
{	color = texture(tex_color, texCoord_TESout);
}
```

```
C++/OpenGL 应用程序
// 这次我们也传入一个纹理以用来绘制表面
// 像往常一样在 init()里加载纹理，并在 display()里启用

void display(GLFWwindow* window, double currentTime) {
	. . .
	glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_2D, textureID);

	glFrontFace(GL_CCW);

	glPatchParameteri(GL_PATCH_VERTICES, 16);	// 每个补丁的顶点数量 = 16
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	glDrawArrays(GL_PATCHES, 0, 16);	// 补丁顶点的总数量：16 x 1 个补丁= 16
}
```

顶点着色器现在指定代表特定贝塞尔曲面的 16 个控制点（“补丁”顶点）。在这个例子中，它们都被归一化到范围[−1...+1]。顶点着色器还使用控制点来确定适合细分网格的纹理坐标，其值在[0...1]范围内。很重要的是，要重申顶点着色器输出的顶点不是将要用来光栅化的顶点，而是贝塞尔控制点。使用曲面细分时，补丁顶点永远不会被光栅化——只有曲面细分顶点会被光栅化。

控制着色器仍然会指定内部和外部曲面细分级别。它现在还负责将控制点和纹理坐标发

送到评估着色器。请注意，曲面细分级别只需要指定一次，因此该步骤仅在第 0 次调用期间完成（回想一下 TCS 每个顶点运行一次，因此在此示例中有 16 次调用）。为方便起见，我们为每个细分级别指定了 32 个细分。

接下来，评估着色器执行所有贝塞尔曲面计算。main()开头的大块赋值语句从每个传入 gl_in 的 gl_Position 中提取控制点（请注意，这些控制点对应于控制着色器的 gl_out 变量）。然后使用来自曲面细分器的网格点计算混合函数的权重，从而生成一个新的 outputPosition，然后应用模型-视图-投影矩阵，为每个网格点生成输出 gl_Position 并形成贝塞尔曲面。

另外，还需要创建纹理坐标。顶点着色器仅为每个控制点位置提供一个纹理坐标。但我们并不是要渲染控制点，我们最终需要更多的曲面细分网格点的纹理坐标。有很多方法可以做到这一点，在这里我们利用 GLSL 方便的混合功能对它们进行线性插值。mix()函数需要 3 个参数：（a）起始点；（b）结束点；（c）内插值，范围为 0～1。它返回与内插值对应的起点和终点之间的值。由于细分网格坐标的范围也是 0～1，所以它们可以直接用于此目的。

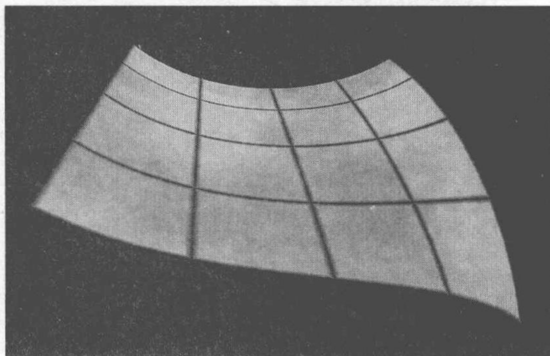


图 12.5 曲面细分过的贝塞尔曲面

这次在片段着色器中，不再是输出单一颜色，而是应用标准纹理。属性 texCoord_TESout 中的纹理坐标是在评估着色器中生成的纹理坐标。对 C++程序的更改同样很简单——请注意，现在指定的补丁大小为 16。结果输出如图 12.5 所示（应用了[LU16]的平铺纹理）。

12.3 地形、高度图的细分

回想一下，在顶点着色器中执行高度贴图可能会遇到顶点数量不足以用来渲染所需的细节的情况。现在我们有了生成大量顶点的方法，让我们回到 Hastings-Trew 的月球表面纹理贴图[HT16]并将其用作高度贴图，提升曲面细分顶点来生成月球表面细节。正如我们将看到的，这具有一些优点，可以让顶点的几何形状更好地匹配月亮图像，并且提升轮廓（边缘）细节。

我们的策略是修改程序 12.1，在 *X-Z* 平面中放置细分网格，并使用高度贴图来设置每个细分网格点的 *Y* 坐标。要做到这一点，我们不需要补丁，因为可以硬编码细分网格的位置，因此我们将在 glDrawArrays()和 glPatchParameteri()中为每个补丁指定所需的最少的 1 个顶点，如程序 12.1 中所做的那样。Hastings-Trew 的月亮纹理图像既用于颜色，也用作高度图。

我们通过将曲面细分网格的 gl_TessCoord 值映射到顶点和纹理的适当范围，在评估着色器中生成顶点和纹理坐标。[①]评估着色器也通过添加月亮纹理的一小部分颜色分量到输出顶

① 在某些应用程序中，纹理坐标是在外部生成的，例如，在使用曲面细分为导入的模型提供额外顶点时。在这种情况下，需要对提供的纹理坐标进行插值。

点的 *Y* 分量，来实现高度贴图。着色器的更改显示在程序 12.3 中。

程序 12.3 简单的地形曲面细分

顶点着色器

```
#version 430
uniform mat4 mvp_matrix;
layout (binding = 0) uniform sampler2D tex_color;
void main(void) { }
```

曲面细分控制着色器

```
. . .
layout (vertices = 1) out;             // 这个应用程序中不需要控制点

void main(void)
{ int TL=32;
  if (gl_InvocationID == 0)
  { gl_TessLevelOuter[0] = TL; gl_TessLevelOuter[2] = TL;
    gl_TessLevelOuter[1] = TL; gl_TessLevelOuter[3] = TL;
    gl_TessLevelInner[0] = TL; gl_TessLevelInner[1] = TL;
  }
}
```

曲面细分评估着色器

```
. . .
out vec2 tes_out;
uniform mat4 mvp_matrix;
layout (binding = 0) uniform sampler2D tex_color;

void main (void)
{ // 将曲面细分网格顶点从[0...1]映射到想要的顶点[-0.5...+0.5]
  vec4 tessellatedPoint = vec4(gl_TessCoord.x - 0.5, 0.0, gl_TessCoord.y - 0.5, 1.0);

  // 垂直"翻转" Y 值，以将曲面细分网格顶点映射到纹理坐标
  // 左上顶点坐标是(0,0)，左下纹理坐标是(0,0)
  vec2 tc = vec2(gl_TessCoord.x, 1.0 - gl_TessCoord.y);

  // 图像是灰度图，所以任何一个颜色分量（R、G 或 B）都可以作为高度偏移量
  tessellatedPoint.y += (texture(tex_color, tc).r) / 40.0;    // 将颜色值等比例缩小应用于 Y 值

  // 将高度贴图提升的点转换到视觉空间
  gl_Position = mvp_matrix * tessellatedPoint;
  tes_out = tc;
}
```

片段着色器

```
. . .
in vec2 tes_out;
out vec4 color;
layout (binding = 0) uniform sampler2D tex_color;

void main(void)
{ color = texture(tex_color, tes_out);
}
```

这里的片段着色器类似于程序 12.2 的，只是根据纹理图像输出颜色。C++/OpenGL 应用程序基本上没有变化——它加载纹理（用作纹理和高度图）并为其启用采样器。图 12.6 显示了纹理图像（左侧）和第一次尝试的最终输出，遗憾的是，它还没有实现正确的高度贴图。

第一次结果存在严重缺陷。虽然我们现在可以看到远处地平线上的轮廓细节，但是那里的凸起与纹理贴图中的实际细节不对应。回想一下，在高度图中，白色应该表示“高”，而黑色应该表示“低”。特别是图像右上方的区域显示的大山丘与其中的浅色和深色无关。

导致此问题的原因是细分网格的分辨率。曲面细分器可以生成的最大顶点数取决于硬件。要符合 OpenGL 标准，唯一的要求是每个曲面细分级别的最大值至少为 64。我们的程序指定了一个内部和外部曲面细分级别均为 32 的单一细分网格，因此我们生成了大约 32×32 或者说刚刚超过 1 000 个顶点，这不足以准确反映图像中的细节。这在图 12.6 右上方（图中放大）尤其明显——边缘细节仅在沿地平线的 32 个点处采样，这会产生巨大而看起来很随机的山丘。即使我们将曲面细分值增加到 64，总共 64×64 或刚刚超过 4 000 个顶点仍然不足以满足使用月球图像进行高度贴图的需要。

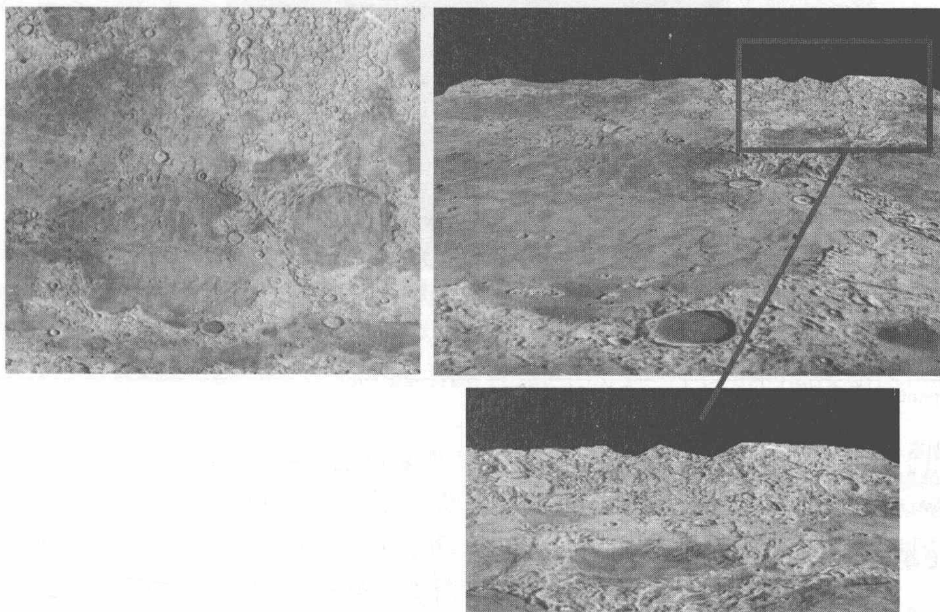


图 12.6 细分地形——首次尝试失败，顶点数量不足

增加顶点数量的一个好方法是使用我们在第 4 章中看到的实例化。我们的策略是让曲面细分器生成网格，并使用实例化重复数次。在顶点着色器中，我们构建了一个由 4 个顶点定义的补丁，每个顶点用于细分网格的每个角。在我们的 C++/OpenGL 应用程序中，我们将 glDrawArrays()调用更改为 glDrawArraysInstanced()。如此，我们指定一个 64×64 个补丁的网格，每个补丁包含一个细分级别为 32 的网格。这将带给我们总共 64×64×32×32 个，或者说超过 400 万个顶点。

顶点着色器首先指定 4 个纹理坐标(0,0)、(0,1)、(1,0)和(1,1)。使用实例化时，请回想一下，顶点着色器可以访问整数变量 gl_InstanceID，它包含一个对应于当前正在处理的 glDrawArraysInstanced()调用的计数器。我们使用此 ID 值来分配大网格中各个补丁的位置。补丁位于行和列中，第一个补丁位于(0,0)，第二个位于(1,0)，下一个位于(2,0)，依此类推，第一列中的最后一个补丁在(63,0)。下一列的补丁位于(0,1)、(1,1)，依此类推，直至(63,1)。最后一列的补丁位于(0,63)、(1,63)，依此类推，最后是(63,63)。给定补丁的 *X* 坐标是实例 ID

整除 64，Y 坐标是实例 ID 除以 64（整数除法）。然后着色器将坐标向下缩放到范围[0...1]。

控制着色器没有更改，除了它将顶点和纹理坐标传递下去。

接下来，评估着色器获取传入的细分网格顶点（由 gl_TessCoord 指定）并将它们移动到传入补丁指定的坐标范围内。它对纹理坐标也进行一样的处理，并且也会以与程序 12.3 中相同的方式应用高度贴图。片段着色器没有修改。

每个组件的更改显示在程序 12.4 中。结果如图 12.7 所示。请注意，高点和低点现在更接近于图像的亮部和暗部。

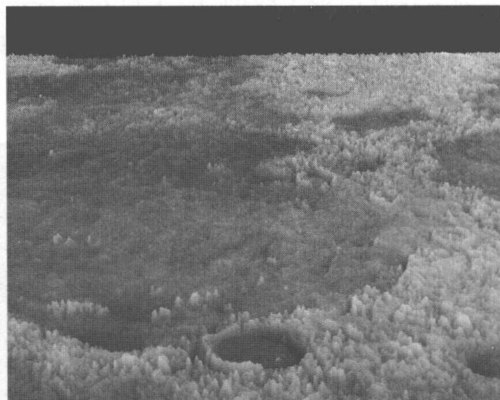


图 12.7 细分地形——第二次尝试，使用实例化

程序 12.4 实例化细分地形

```
C++/OpenGL 应用程序

// 和贝塞尔曲面例子相同，并做如下修改
glPatchParameteri(GL_PATCH_VERTICES, 4);
glDrawArraysInstanced(GL_PATCHES, 0, 4, 64*64);

顶点着色器
. . .
out vec2 tc;

void main(void)
{ vec2 patchTexCoords[ ] = vec2[ ] (vec2(0,0), vec2(1,0), vec2(0,1), vec2(1,1));

  // 基于当前是哪个实例计算出坐标偏移量
  int x = gl_InstanceID % 64;
  int y = gl_InstanceID / 64;

  // 纹理坐标被分配进 64 个补丁中，并归一化到[0..1]。翻转 Y 轴坐标
  tc = vec2( (x+patchTexCoords[gl_VertexID].x) / 64.0, (63 - y+patchTexCoords[gl_VertexID].y) / 64.0);

  // 顶点位置和纹理坐标相同，只是它的取值范围从-0.5 到+0.5
  gl_Position = vec4(tc.x - 0.5, 0.0, (1.0 - tc.y) - 0.5, 1.0);          // 并且将 Y 轴坐标翻转回来
}

曲面细分控制着色器
. . .
layout (vertices = 4) out;
in vec2 tc[ ];
out vec2 tcs_out[ ];
```

```
void main(void)
{ // 曲面细分级别的指定和之前例子中相同
  . . .
  tcs_out[gl_InvocationID] = tc[gl_InvocationID];
  gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

曲面细分评估着色器

```
. . .
in vec2 tcs_out[ ];
out vec2 tes_out;
void main (void)
{ // 将纹理坐标映射到传入的控制点指定的子网格上
  vec2 tc = vec2(tcs_out[0].x + (gl_TessCoord.x) / 64.0, tcs_out[0].y + (1.0 - gl_TessCoord.y) / 64.0);

  // 将细分网格映射到传入的控制点指定的子网格上
  vec4 tessellatedPoint = vec4(gl_in[0].gl_Position.x + gl_TessCoord.x / 64.0, 0.0,
                               gl_in[0].gl_Position.z + gl_TessCoord.y / 64.0, 1.0);

  // 将高度图的高度增加给顶点
  tessellatedPoint.y += (texture(tex_height, tc).r) / 40.0;
  gl_Position = mvp_matrix * tessellatedPoint;
  tes_out = tc;
}
```

现在我们已经实现了高度贴图，我们可以着手改进它并整合光照。一个挑战是我们的顶点还没有与它们相关的法向量。另一个挑战是简单地使用纹理图像作为高度图产生了过度“锯齿状”的结果——在这种情况下是因为并非纹理图像中的所有灰度变化都是由高度引起的。对于这个特定的纹理贴图，Hastings-Trew 已经生成了一个改进的高度贴图，我们可以使用[HT16]。如图 12.8 左图所示。

我们可以通过生成相邻顶点（或高度图中的相邻纹素）的高度，构建连接它们的向量以及使用叉积来计算法向量，以动态计算和创建法向量。这需要一些细微的调整，具体取决于场景的精度（和/或高度图图像）。在这里，我们使用 GIMP “normalmap” 插件[GP16]来根据 Hastings-Trew 的高度图生成法线贴图，如图 12.8 右图所示。

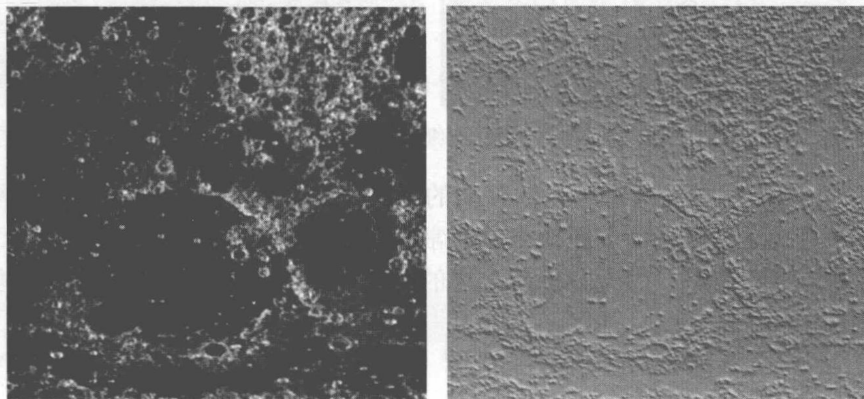


图 12.8 月球表面：高度图[HT16]（左）和法线贴图（右）

我们对代码进行的大部分更改现在只是为了实现 Phong 着色的标准方法。

- C++/OpenGL 应用程序。

我们加载并激活一个额外的纹理来保存法线贴图，还添加了代码来指定光照和材质，就像我们在以前的应用程序中所做的那样。

- 顶点着色器。

唯一的增补是光照统一变量的声明和法线贴图的采样器。通常在顶点着色器中完成的光照代码被移动到曲面细分评估着色器，因为直到曲面细分阶段才生成顶点。

- 曲面细分控制着色器。

唯一的增补是光照统一变量的声明和法线贴图的采样器。

- 曲面细分评估着色器。

Phong 光照的准备代码现在放在评估着色器中：

```
varyingVertPos = (mv_matrix * position).xyz;
varyingLightDir = light.position - varyingVertPos;
```

- 片段着色器。

这里完成了用于计算 Phong（或 Blinn-Phong）照明的典型代码段，以及从法线贴图中提取法向量的代码。然后将光照结果与纹理图像用加权求和的方式结合起来。

带有高度和法线贴图以及 Phong 照明的最终结果如图 12.9 所示。地形现在会响应光照。在此示例中，位置光已放置在左侧图像中心的左侧，右侧图像中心的右侧。

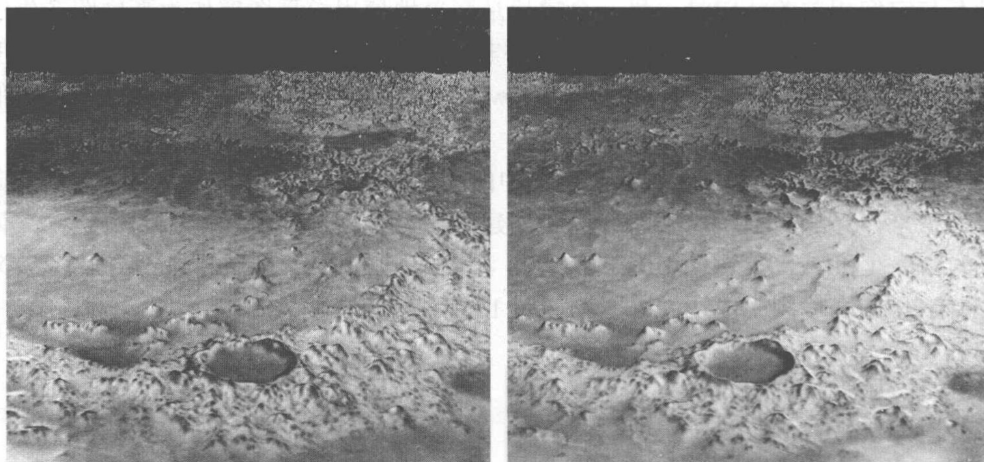


图 12.9 具有法线贴图和光照的曲面细分地形（光源分别位于左侧和右侧）

尽管从静止图像很难判断出对光的移动的响应，但是读者应该能够辨别出漫射光的变化，并且山峰的镜面高光在两个图像中是非常不同的。当摄像机或光源移动时，这当然会更明显。结果仍然不完美，因为无论什么样的光照，输出中包含的原始纹理都包括了将出现在渲染结果上的阴影。

12.4 控制细节级别（LOD）

在程序 12.4 中，使用实例化来实时生成数百万个顶点，即使是装备精良的现代计算机也

可能会感受到负担。幸运的是，将地形划分为单独的补丁的策略，正如我们为增加生成的网格顶点的数量所做的那样，也为我们提供了一种减少负担的好机制。

在生成的数百万个顶点中，许多顶点不是必需的。靠近摄像机的补丁中的顶点非常重要，因为我们希望能够识别附近物体的细节。但是，补丁越远离摄像机，甚至光栅化过程中有足够的像素来体现我们生成的顶点数量的可能性就越小！

根据距摄像机的距离更改补丁中的顶点数量是一种称为**细节级别**或 LOD 的技术。Sellers 等人描述了一种通过修改控制着色器来控制实例化曲面细分中的 LOD 的方法[SW15]。程序 12.5 显示了 Sellers 等人的方法的简化版本。策略是使用补丁的感知大小来确定其曲面细分级别的值。由于补丁的细分网格最终将放置在由进入控制着色器的 4 个控制点定义的方格内，我们可以使用控制点相对于摄像机的位置来确定应该为补丁生成多少个顶点。其步骤如下。

（1）通过将 MVP 矩阵应用于 4 个控制点，计算它们的屏幕位置。

（2）计算由控制点（在屏幕上的空间中）定义的正方形边长（即宽度和高度）。请注意，即使 4 个控制点形成正方形，这些边长也可能不同，因为应用了透视矩阵。

（3）根据曲面细分级别所需的精度（基于高度图中的细节数量），将长度的值按可调整常数进行缩放。

（4）将缩放长度值加 1，以避免将曲面细分级别指定为 0（这将导致不生成顶点）。

（5）将曲面细分级别设置为相应的计算宽度和高度值。

回想一下，在我们的实例中，我们不是只创建一个网格，而是创建 64×64 个网格。因此，对每个补丁执行以上列表中的 5 个步骤，细节级别因补丁而异。

所有更改都在控制着色器中，并显示在程序 12.5 中，生成的输出如图 12.10 所示。请注意，变量 gl_InvocationID 指的是正在处理补丁中的哪个顶点（而不是正在处理哪个补丁）。因此，告诉曲面细分器在每个补丁中生成多少个顶点的 LOD 计算发生在每个补丁的第 0 个顶点期间。

程序 12.5 曲面细分细节级别（LOD）

曲面细分控制着色器

```
. . .
void main(void)
{ float subdivisions = 16.0;            // 基于高度图中细节密度的可调整的常量
  if (gl_InvocationID == 0)
  { vec4 p0 = mvp * gl_in[0].gl_Position;          // 屏幕空间中控制点的位置
    vec4 p1 = mvp * gl_in[1].gl_Position;
    vec4 p2 = mvp * gl_in[2].gl_Position;
    p0 = p0 / p0.w;
    p1 = p1 / p1.w;
    p2 = p2 / p2.w;
    float width = length(p2.xy - p0.xy) * subdivisions + 1.0;      // 曲面细分网格的感知"宽度"
    float height = length(p1.xy - p0.xy) * subdivisions + 1.0;     // 曲面细分网格的感知"高度"
    gl_TessLevelOuter[0] = height;           // 基于感知的边长设置曲面细分级别
    gl_TessLevelOuter[1] = width;
    gl_TessLevelOuter[2] = height;
    gl_TessLevelOuter[3] = width;
    gl_TessLevelInner[0] = width;
    gl_TessLevelInner[1] = height;
  }
  //  像以前一样将纹理坐标和控制点发送给 TES
```

```
    tcs_out[gl_InvocationID] = tc[gl_InvocationID];
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
```

将这些控制着色器的更改应用于图 12.7 中我们场景的实例化（但不带光照）版本，并将高度图替换为 Hastings-Trew 的更精细调整的版本（如图 12.8 所示），将会生成改善的场景，带有更逼真的地平线细节（如图 12.10 所示）。

在此示例中，更改评估着色器中的布局说明符也很有用：

```
layout (quads, equal_spacing) in;
```

更改为：

```
layout (quads, fractional_even_spacing) in;
```

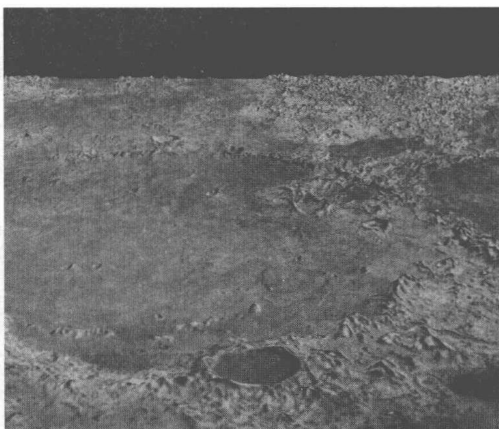


图 12.10 具有控制细节级别（LOD）的曲面细分月亮

在静止图像中难以说明这种修改的原因。在动画场景中，当曲面细分对象在 3D 空间中移动时，如果使用 LOD，有时可以在对象表面上看到曲面细分级别的变化，看起来像一种叫作"弹出"的摆动伪影。从等间距变为分数间距，通过使相邻补丁实例的网格几何体更相似，达成了即使它们的细节级别不同，也可以减少此影响的目的。（参见习题 12.2 和 12.3。）

使用 LOD 可以显著降低系统负载。例如，在动画时，如果不控制 LOD，场景可能会出现不稳定或滞后的情况。

将这种简单的 LOD 技术应用于包含 Phong 着色的版本（程序 12.4）有点棘手。这是因为相邻补丁实例之间的 LOD 变化反过来会导致相关法向量的突然变化，从而导致光照中的弹出伪影！与以往一样，在构建复杂的 3D 场景时需要权衡和妥协。

补充说明

将曲面细分与 LOD 组合在实时虚拟现实应用中特别有用，例如在计算机游戏中，其需要复杂的现实主义细节和频繁的物体移动和/或摄像机位置的变化。在本章中，我们已经说明了曲面细分和 LOD 用于实时地形生成的应用场景，尽管它也可以应用于其他领域，例如 3D 模型的位移贴图（曲面细分顶点被添加到模型的表面，然后被移动以便添加细节）在计

算机辅助设计应用程序中也很有用。

Sellers 等人通过消除摄像机后方的补丁中的顶点（他们通过将内部和外部级别设置为零来实现这一点）[SW15]，进一步扩展了 LOD 技术（在程序 12.5 中显示）。这是一个剔除技术的示例，是一项非常有用的技术，因为实例化细分的负载仍然可以在系统上正常运行。

程序 12.1 中描述的 createShaderProgram()的 4 参数版本被添加到 Utils.cpp 文件中。稍后，我们将添加其他版本以适应几何着色器阶段。

习题

12.1　修改程序 12.1 以试验内部和外部曲面细分级别的各种值，并观察生成的渲染网格。

12.2　修改程序 12.1，将评估着色器中的布局说明符从 equal_spacing 更改为 fractional_even_spacing，如第 12.4 节所述。观察对生成的网格的影响。

12.3　测试程序 12.5，将评估着色器中的布局说明符设置为 equal_spacing，然后设置为 fractional_even_spacing，如第 12.4 节所述。在摄像机移动时观察渲染表面上的效果。您应该能够在第一种情况下观察弹出伪影，这在第二种情况下大多得到缓解。

12.4　（项目）修改程序 12.3 以使用自己设计的高度图（可以使用之前在习题 10.2 中构建的高度图）。然后添加光照和阴影贴图，以便细分地形投射阴影。这是一个复杂的练习，因为第一个和第二个阴影贴图过程中的某些代码需要被移动到评估着色器中。

参考资料

[GP16] GIMP Plugin Registry, normalmap plugin, accessed October 2018.

[HT16] J. Hastings-Trew, JHT's Planetary Pixel Emporium, accessed October 2018.

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7th ed. (Addison-Wesley, 2015).

[TS16] Tessellation, Wikipedia, accessed October 2018.

第13章 几何着色器

在 OpenGL 管线中，紧跟着曲面细分阶段的是几何阶段。在这一阶段中，程序员可以选择包含几何着色器。这个阶段实际上在曲面细分阶段出现之前就已经存在，它在 3.2 版本（2009 年）成为 OpenGL 核心的一部分。

与曲面细分一样，几何着色器使程序员能够以顶点着色器中无法实现的方式操纵顶点组。在某些情况下，可以使用曲面细分着色器或者几何着色器完成同样的任务，因为它们的功能在某些方面重叠。

13.1 OpenGL 中的逐个图元处理

几何着色器阶段位于曲面细分和光栅化之间，位于用于图元处理的管线段内（见图 2.2）。顶点着色器允许一次操作一个顶点，而片段着色器一次可以操作一个片段（实际上是一个像素），但几何着色器却可以一次操作一个图元。

回想一下，图元是 OpenGL 中绘制对象的基本元件。只有少数几种类型的图元；我们将主要关注操纵三角形图元的几何着色器。因此，当我们说几何着色器可以一次操作一个图元时，我们通常意味着着色器一次可以访问三角形的 3 个顶点。几何着色器允许一次性访问图元中的所有顶点，然后：

- 输出相同的图元保持不变；
- 输出修改了顶点位置的相同类型图元；
- 输出不同类型的图元；
- 输出更多的其他图元；
- 删除图元（根本不输出）。

与曲面细分评估着色器类似，可以在几何着色器中将传入的顶点属性作为数组进行访问。但是，在几何着色器中，传入属性数组仅索引到图元尺寸那么大。例如，如果图元是三角形，则可用索引为 0、1、2。使用预先定义的数组 gl_in 访问顶点数据本身，如下所示。

```
gl_in[2].gl_Position          // 第三个顶点的位置
```

与曲面细分评估着色器类似，几何着色器输出的顶点属性都是标量。也就是说，输出是形成图元的各个顶点（它们的位置和其他属性变量，如果有的话）的流。

有一个布局修饰符用于设置图元输入/输出类型和输出大小。特殊的 GLSL 命令 EmitVertex()指定了将要输出一个顶点。特殊的 GLSL 命令 EndPrimitive()表示一个特定的图元构建完成。

有一个内置变量 gl_PrimitiveIDIn，它保存当前图元的 ID。ID 从 0 开始，并计数到图元

总数减 1。

我们将探讨四种常见的操作类型：

- 修改图元；
- 删除图元；
- 添加图元；
- 更改图元类型。

13.2 修改图元

当通过对图元（通常为三角形）的单独更改就可以影响对象形状的改变时，使用几何着色器就很方便。

例如，考虑我们之前在图 7.12 中呈现的环面。假设环面代表内部的空间（例如当表示轮胎时），而我们想要给它“充气”。简单地在 C++/OpenGL 代码中应用比例缩放因子将无法实现这一点，因为它的基本形状不会改变。想要让其显示出“充气”的外观，还需要在环面伸入空的中心空间时使内孔变小。

解决这个问题的一种方法是将表面法向量添加到每个顶点。虽然这可以在顶点着色器中完成，但是我们在几何着色器中进行练习。程序 13.1 显示了 GLSL 几何着色器的代码。其他模块与程序 7.3 相同，只有一些小改动：片段着色器输入名称现在需要反映几何着色器的输出（例如，varyingNormal 变为 varyingNormalG），C++/OpenGL 应用程序需要编译几何着色器并在链接之前将其附加到着色器程序。新着色器被指定为几何着色器，如下所示。

```
GLuint gShader = glCreateShader(GL_GEOMETRY_SHADER);
```

程序 13.1 几何着色器：修改顶点

```
#version 430

layout (triangles) in;

in vec3 varyingNormal[ ];          // 来自顶点着色器的输入
in vec3 varyingLightDir[ ];
in vec3 varyingHalfVector[ ];

out vec3 varyingNormalG;           // 输出给光栅着色器然后到片段着色器
out vec3 varyingLightDirG;
out vec3 varyingHalfVectorG;

layout (triangle_strip, max_vertices=3) out;

// 矩阵和光照统一变量和以前一样
. . .
void main (void)
{ // 沿着法向量移动顶点，并将其他顶点属性原样传递
  for (int i=0; i<3; i++)
  { gl_Position = proj_matrix *
      gl_in[i].gl_Position + normalize(vec4(varyingNormal[i],1.0)) * 0.4;
    varyingNormalG = varyingNormal[i];
```

```
    varyingLightDirG = varyingLightDir[i];
    varyingHalfVectorG = varyingHalfVector[i];
    EmitVertex();
  }
  EndPrimitive();
}
```

在程序 13.1 中需要注意，与顶点着色器的输出变量对应的输入变量被声明为数组。这为程序员提供了一种机制，可以使用索引 0、1 和 2 访问三角形图元中的每个顶点及其属性。我们希望沿着它们的表面法向量向外移动这些顶点。在顶点着色器中，顶点和法向量都已经被转换到视图空间。我们为每个传入的顶点位置（gl_in[i].gl_Position）添加法向量的一小部分，然后将投影矩阵应用于结果，生成每个输出 gl_Position。

值得注意的是，使用 GLSL 调用 EmitVertex()来指定我们何时完成了计算输出 gl_Position 及其相关的顶点属性并准备输出顶点。EndPrimitive()调用指定我们已经完成了组成图元（在本例中为三角形）的一组顶点的定义。结果如图 13.1 所示。

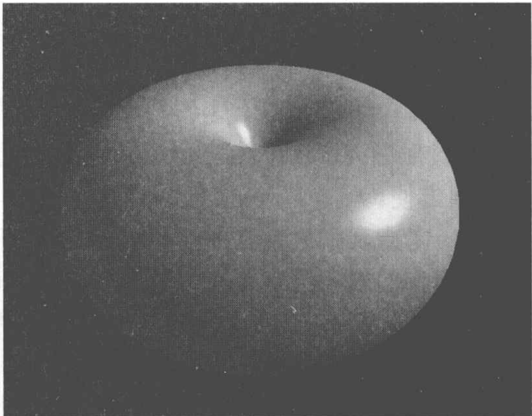


图 13.1 “充气”的环面，顶点由几何着色器修改

几何着色器包括两个布局限定符。第一个指定输入图元类型，并且必须与 C++端 glDrawArrays()或 glDrawElements()调用中的图元类型兼容。选项如表 13.1 所示。

表 13.1 图元输入类型的选项

几何着色器输入图元类型	与 glDrawArrays()调用兼容的图元类型	每次调用顶点的数量
points	GL_POINTS	1
lines	GL_LINES, GL_LINE_STRIP	2
lines_adjacency	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY	4
triangles	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN	3
triangles_adjacency	CL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY	6

各种 OpenGL 图元类型（包括“strip”和“fan”类型）在第 4 章中讲过。“相邻”类型在 OpenGL 中用来与几何着色器一起使用，并且它们可以访问与图元相邻的顶点。我们在本书中不使用它们，但为了完整性，依然列出它们。

输出图元类型必须是 points、line_strip 或 triangle_strip。请注意，输出布局限定符也会指定着色器在每次调用中输出的最大顶点数。

在顶点着色器中可以更容易地对环面进行这种特定的改变。然而，假设不是沿着自己的表面法向量向外移动每个顶点，而是希望将每个三角形沿其表面法向量向外移动，实际上

是将环面的组成三角形向外“爆炸”。顶点着色器做不到这一点，因为计算三角形的法向量需要对 3 个三角形顶点的顶点法向量进行平均，并且顶点着色器一次只能访问三角形中一个顶点的顶点属性。但是，我们可以在几何着色器中执行此操作，因为几何着色器可以访问每个三角形中的所有 3 个顶点。我们平均它们的法向量来计算三角形的曲面法向量，然后将该平均法向量加给三角形图元中的每个顶点。图 13.2、图 13.3 和图 13.4 分别显示了曲面法向量的平均值、修改后的几何着色器 main()代码和输出的结果。

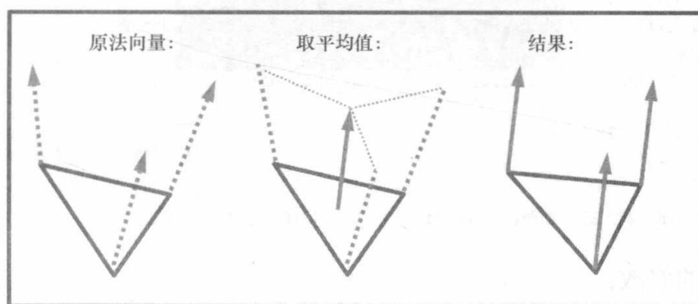


图 13.2 将平均三角形曲面法向量应用于三角形顶点

```
void main (void)
{   // 对三角形3个顶点法向量取平均值，得到三角形的曲面法向量
    vec4 triangleNormal =
       vec4(((varyingNormal[0] + varyingNormal[1] + varyingNormal[2]) / 3.0),1.0);

    //  将三个点都沿所得法向量移动
    for (i=0; i<3; i++)
    {   gl_Position = proj_matrix * (gl_in[i].gl_Position + normalize(triangleNormal) * 0.4);
        varyingNormalG = varyingNormal[i];
        varyingLightDirG = varyingLightDir[i];
        varyingHalfVectorG = varyingHalfVector[i];
        EmitVertex();
    }
    EndPrimitive();
}
```

图 13.3 修改了几何着色器，用于“爆炸”环面

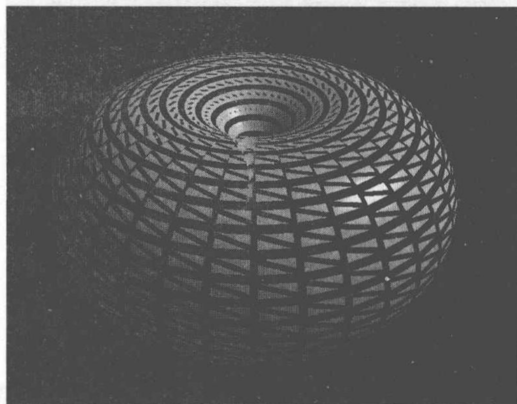


图 13.4 “爆炸”的环面

通过确保环面的内部也是可见的（通常这些三角形会被 OpenGL 剔除，因为它们是“背

面"），可以改善"爆炸"环面的外观。一种解决方式是使环面被渲染两次，一次以正常方式进行，一次使缠绕顺序反转（使缠绕顺序反转实际上相当于切换哪些面朝向前方，哪些面朝向后方）。我们还向着色器（通过统一变量）发送一个标志，以禁用背向三角形上的漫反射和镜面光，以使它们不那么突出。代码的更改如下。

对 display()函数的修改：

```
. . .
// 绘制前向三角形——启用光照
glUniform1i(lLoc, 1);      // 用来启用、禁用漫反射、镜面光组件的统一变量的位置
glFrontFace(GL_CCW);
glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);

// 绘制后向三角形——禁用光照
glUniform1i(lLoc, 0);
glFrontFace(GL_CW);
glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
```

对片段着色器的修改：

```
. . .
if (enableLighting == 1)
{ fragColor = …           // 当渲染前向表面时，使用正常的光照计算
}
else                      // 当渲染后向表面时，只启用环境光照组件
{ fragColor = globalAmbient * material.ambient + light.ambient * material.ambient;
}
```

由此产生的"爆炸"环面，包括背面，如图 13.5 所示。

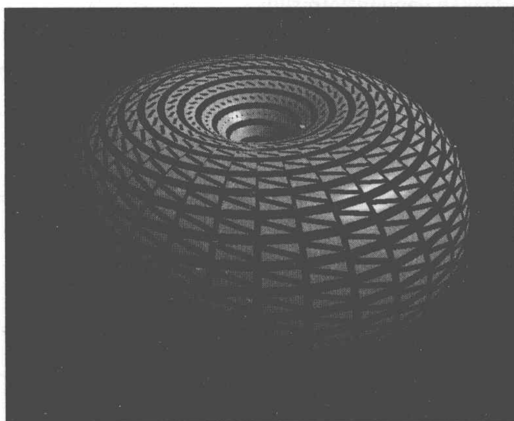


图 13.5 "爆炸"的环面，包括背面

13.3 删除图元

几何着色器的一个常见用途是通过合理地删除一些图元来从简单的对象构建丰富的装饰对象。例如，从我们的环面中移除一些三角形可以将其变成一种复杂的格子结构，而从零开始建模这个结构是更加困难的。执行此操作的几何着色器显示在程序 13.2 中，输出如

图 13.6 所示。

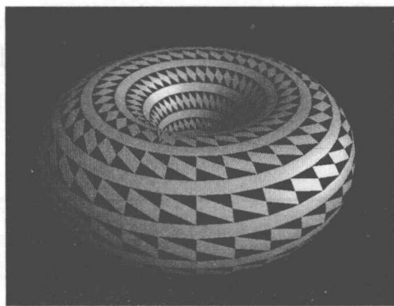


图 13.6 几何着色器：删除图元

程序 13.2 几何着色器：删除图元

```
// 输入、输出和统一变量和以前一样

. . .
void main (void)
{ if ( mod(gl_PrimitiveIDIn,3) != 0 )
  {     for (int i=0; i<3; i++)
        {   gl_Position = proj_matrix * gl_in[i].gl_Position;
            varyingNormalG = varyingNormal[i];
            varyingLightDirG = varyingLightDir[i];
            varyingHalfVectorG = varyingHalfVector[i];
            EmitVertex();
        }
  }
  EndPrimitive();
}
```

不需要对代码进行其他更改。请注意这里使用了 mod 函数——所有顶点，除了每 3 个图元中的第一个图元的顶点被忽略之外，都被传递。在这里，渲染背向三角形也可以提高真实感，如图 13.7 所示。

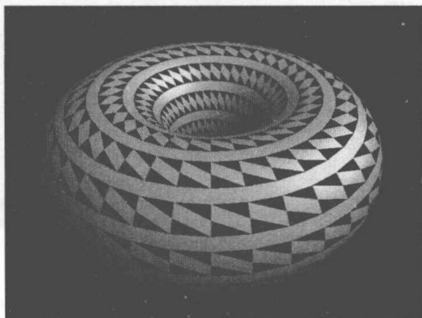


图 13.7 显示背面的图元删除

13.4 添加图元

也许几何着色器最有趣和最有用的用途是为正在渲染的模型添加额外的顶点和/或图元。这使得可以进行诸如增加对象中的细节以改善高度贴图，或者完全改变对象的形状之类的事情。

考虑以下示例，我们将环面中的每个三角形更改为一个微小的三角形金字塔。

我们的策略类似于我们之前的“爆炸”环面示例，如图 13.8 所示。传入三角形图元的顶点用于定义金字塔的基座。金字塔的壁由那些顶点和通过平均原始顶点的法向量计算的新点（称为“尖峰点”）构成。然后通过从尖峰点到基座的两个向量的叉积计算金字塔的 3 个“边”中的每一个的新法向量。

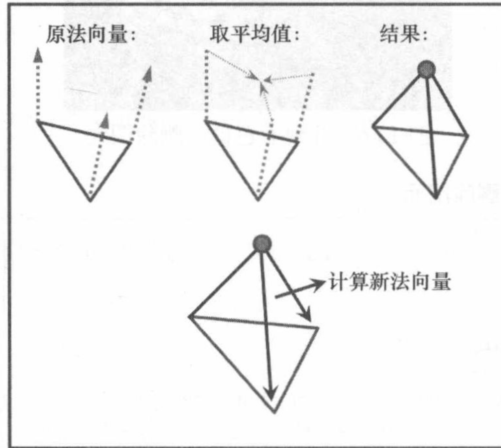


图 13.8 将三角形转换为金字塔

程序 13.3 中的几何着色器为环面中的每个三角形图元执行此操作。对于每个输入三角形，它输出 3 个三角形图元，总共 9 个顶点。每个新三角形都在函数 makeNewTriangle()中构建，该函数被调用 3 次。它计算指定三角形的法向量，然后调用函数 setOutputValues()为发出的每个顶点分配适当的输出顶点属性。在发出所有 3 个顶点之后，它调用 EndPrimitive()。为了确保准确地执行光照，为每个新创建的顶点计算光照方向向量的新值。

程序 13.3 几何着色器：添加图元

```
. . .
vec3 newPoints[ ], lightDir[ ];
float sLen = 0.01;      // sLen 是"尖峰长度", 小金字塔的高度

void setOutputValues(int p, vec3 norm)
{ varyingNormal = norm;
  varyingLightDir = lightDir[p];
  varyingVertPos = newPoints[p];
  gl_Position = proj_matrix * vec4(newPoints[p], 1.0);
}

void makeNewTriangle(int p1, int p2)
{  // 为这个三角形生成表面法向量
   vec3 c1 = normalize(newPoints[p1] - newPoints[3]);
   vec3 c2 = normalize(newPoints[p2] - newPoints[3]);
   vec3 norm = cross(c1,c2);

   // 生成并发出 3 个顶点
   setOutputValues(p1, norm); EmitVertex();
   setOutputValues(p2, norm); EmitVertex();
   setOutputValues(3, norm); EmitVertex();
   EndPrimitive();
```

```
}

void main(void)
{ // 给三个三角形顶点加上原始表面法向量
  vec3 sp0 = gl_in[0].gl_Position.xyz + varyingOriginalNormal[0]*sLen;
  vec3 sp1 = gl_in[1].gl_Position.xyz + varyingOriginalNormal[1]*sLen;
  vec3 sp2 = gl_in[2].gl_Position.xyz + varyingOriginalNormal[2]*sLen;

  // 计算组成小金字塔的新点
  newPoints[0] = gl_in[0].gl_Position.xyz;
  newPoints[1] = gl_in[1].gl_Position.xyz;
  newPoints[2] = gl_in[2].gl_Position.xyz;
  newPoints[3] = (sp0 + sp1 + sp2)/3.0; // 尖峰点

  // 计算从顶点到光照的方向
  lightDir[0] = light.position - newPoints[0];
  lightDir[1] = light.position - newPoints[1];
  lightDir[2] = light.position - newPoints[2];
  lightDir[3] = light.position - newPoints[3];

  // 构建 3 个三角形，以组成小金字塔的表面
  makeNewTriangle(0,1);        // 第三个点永远是尖峰点
  makeNewTriangle(1,2);
  makeNewTriangle(2,0);
}
```

结果输出如图 13.9 所示。如果尖峰长度（sLen）变量增加，则添加的表面“金字塔”将更高。然而，在没有阴影的情况下，它们可能看起来并不真实。将阴影贴图添加到程序 13.3 留作练习。

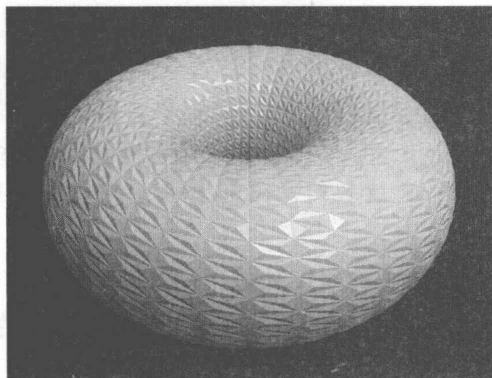


图 13.9 几何着色器：添加图元

仔细应用这种技术可以模拟尖峰、荆棘和其他精细表面突起，或者反向的压痕、凹坑（参考资料[DV14, TR13, KS16]）等。

13.5 更改图元类型

OpenGL 允许在几何着色器中更改图元类型。此功能的一个常见用途是将输入三角形转换为一个或多个输出线段，来模拟毛发或头发。虽然生成令人信服的头发仍然是更难的现

实世界项目之一，但几何着色器可以在许多情况下帮助实现实时渲染。

程序 13.4 显示了一个几何着色器，它将每个输入的 3 个顶点的三角形转换为一个向外的两个顶点的线段。它首先通过平均三角形顶点位置生成三角形的质心，来计算头发束的起点。然后它使用和程序 13.3 中相同的“尖峰点”作为头发的终点。输出图元被指定为具有两个顶点的线段，第一个顶点是起点，第二个顶点是终点。结果显示在图 13.10 中，用于实例化维数为 72 个切片的环面。

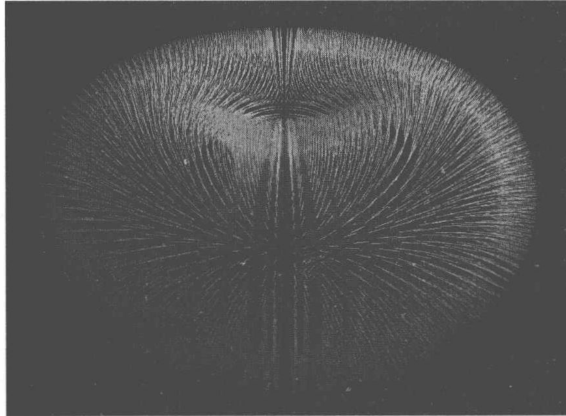


图 13.10 将三角形图元更改为线图元

当然，这仅仅是产生完全逼真头发的起点。使头发弯曲或移动将需要若干修改，例如为线条生成更多顶点并沿曲线计算它们的位置和/或结合随机性。由于线段没有明显的表面法向量，光照会很复杂；在这个例子中，我们简单地指定法向量与原始三角形的表面法向量相同。

程序 13.4 几何着色器：改变图元类型

```
layout (line_strip, max_vertices=2) out;
. . .
void main(void)
{ vec3 op0 = gl_in[0].gl_Position.xyz;                                    // 原始三角形顶点
  vec3 op1 = gl_in[1].gl_Position.xyz;
  vec3 op2 = gl_in[2].gl_Position.xyz;
  vec3 ep0 = gl_in[0].gl_Position.xyz + varyingNormal[0]*sLen;            // 偏移三角形顶点
  vec3 ep1 = gl_in[1].gl_Position.xyz + varyingNormal[1]*sLen;
  vec3 ep2 = gl_in[2].gl_Position.xyz + varyingNormal[2]*sLen;

  // 计算组成小线段的新点
  vec3 newPoint1 = (op0 + op1 + op2)/3.0;                 // 原始（起始）点
  vec3 newPoint2 = (ep0 + ep1 + ep2)/3.0;                 // 结束点

  gl_Position = proj_matrix * vec4(newPoint1, 1.0);
  varyingVertPosG = newPoint1;
  varyingLightDirG = light.position - newPoint1;
  varyingNormalG = varyingNormal[0];
  EmitVertex();

  gl_Position = proj_matrix * vec4(newPoint2, 1.0);
  varyingVertPosG = newPoint2;
  varyingLightDirG = light.position - newPoint2;
  varyingNormalG = varyingNormal[1];
```

```
    EmitVertex();

    EndPrimitive();
}
```

补充说明

几何着色器吸引人的一点在于它们相对容易使用。虽然几何着色器的许多应用可以使用曲面细分来实现，但几何着色器的机制通常使它们更容易实现和调试。当然，几何与曲面细分的相对适用范围取决于特定的应用。

生成令人信服的真实头发或毛发具有挑战性，并且根据应用场景需要采用多种技术。在某些情况下，简单的纹理就足够了，或者可以使用曲面细分或几何着色器，例如本章所示的基本技术。当需要更真实的效果时，移动（动画）和光照变得棘手。头发和毛发生成的两个专用工具是 HairWorks 和 TressFX。HairWorks 是 NVIDIA GameWorks 套件[GW18]的一部分，而 TressFX 是由 AMD 开发的[TR18]。前者适用于 OpenGL 和 DirectX，而后者仅适用于 DirectX。使用 TressFX 的例子可以在[GP14]中找到。

习题

13.1　修改程序 13.1，使其将每个顶点略微移向其原始三角形的中心。结果应该类似于图 13.5 中的爆炸环面，但没有环面尺寸的整体变化。

13.2　修改程序 13.2，使其删除每第 2 个图元或每第 4 个图元（而不是每第 3 个图元），并观察对生成的渲染环面的影响。此外，尝试将实例化环面的维度更改为不是 3 的倍数的值（例如 40），同时仍然删除每第 3 个图元，这会有许多可能的影响。

13.3　（项目）修改程序 13.4 以额外渲染原始环面。也就是说，渲染一个有光照的环面（如前面第 7 章所述）和向外线段（使用几何着色器），使“头发”看起来像是从环面中出来的。

13.4　（研究和项目）修改程序 13.4，使其生成具有两个以上顶点的向外线段，这些顶点排列使得线段看起来略微弯曲。

参考资料

[DV14] J. deVries, LearnOpenGL, 2014, accessed October 2018.

[GP14] *GPU Pro 5: Advanced Rendering Techniques*, ed. W. Engel (CRC Press, 2014).

[GW18] NVIDIA GameWorks Suite, 2018, accessed May 2018.

[KS16] J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9th ed. (Addison-Wesley, 2016).

[TR13] P. Trettner, “Prototype Grass” (blog), 2013, accessed October 2018.

[TR18] TressFX Hair, AMD, 2018, accessed May 2018.

第 14 章　其他技术

在本章中，我们将使用在本书中学到的工具来探索各种技术。有些我们会完全讲解，而其他一些我们将只会粗略描述。图形编程是一个巨大的领域，本章绝不是全面的，而是介绍了多年来发展的一些创造性效果。

14.1　雾

通常当人们想到雾时，他们会想到有雾的早晨，能见度很低。事实上，大气雾霾（如雾）比我们大多数人认为的更常见。大多数时候，空气中都会有一定程度的雾霾，我们已经习惯于看到它，通常不会意识到它的存在，所以我们可以通过引入雾来增强我们室外场景的真实感——即使只是少量。

雾也可以增强深度感。近处物体比远处物体具有更高的清晰度，对于我们的大脑是可以用来破译 3D 场景的地形结构的另一个视觉提示。

模拟雾的方法有很多种，从非常简单的模型到包含光散射效应的复杂模型。即使非常简单的方法也是有效的。有一种方法是基于物体距眼睛的距离将实际像素颜色与另一种颜色（“雾”的颜色通常是灰色或蓝灰色——也用于背景颜色）混合。

图 14.1（见彩插）说明了这个概念。眼睛（相机）显示在左侧，两个红色物体放置在视锥体中。圆柱体更靠近眼睛，所以它主要是原始颜色（红色）；立方体远离眼睛，所以它主要是雾色。对于这个简单的实现，几乎所有的计算都可以在片段着色器中执行。

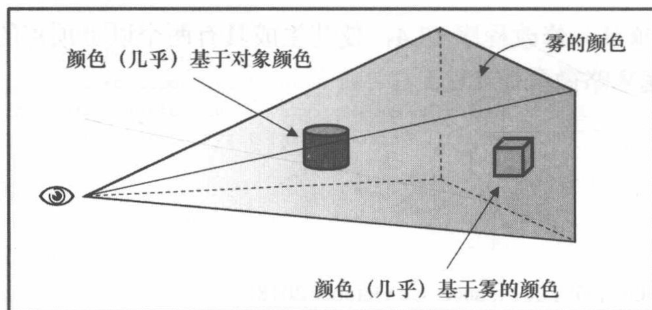


图 14.1　雾：基于距离的混合

程序 14.1 显示了一个非常简单的雾算法的相关代码，该算法按照从相机到像素的距离，使用从对象颜色到雾颜色的线性混合。具体来说，此示例将雾添加到程序 10.4 中的高度贴图示例。

程序 14.1 简单的雾生成

```
顶点着色器

. . .
out vec3 vertEyeSpacePos;
. . .
// 在视觉空间中不考虑透视计算顶点位置，并将它发送给片段着色器
// 变量"p"是高度贴图后的顶点，正如程序 10.4 中所述
vertEyeSpacePos = (mv_matrix * p).xyz;
片段着色器
. . .
in vec3 vertEyeSpacePos;
out vec4 fragColor;
. . .
void main(void)
{ vec4 fogColor = vec4(0.7, 0.8, 0.9, 1.0);
  float fogStart = 0.2;
  float fogEnd = 0.8;

  // 在视觉空间中从摄像机到顶点的距离就是到这个顶点的向量的长度，因为摄像机在视觉空间中的(0,0,0)位置
  float dist = length(vertEyeSpace.xyz);
  float fogFactor = clamp(((fogEnd - dist) / (fogEnd - fogStart)), 0.0, 1.0);
  fragColor = mix(fogColor, (texture(t,tc), fogFactor);
}
```

变量 fogColor 指定雾的颜色。变量 fogStart 和 fogEnd 指定输出颜色从对象颜色过渡到雾色的范围（在视觉空间中），并且可以调整以满足场景的需要。在对象颜色中混合的雾的百分比在变量 fogFactor 中计算，该变量是顶点与 fogEnd 的接近程度与过渡区域的总长度之比。GLSL 的 clamp()函数用于将此比率限制在值 0.0 和 1.0 之间。然后，GLSL 的 mix()函数根据 fogFactor 的值返回雾颜色和对象颜色的加权平均值。图 14.2（见彩插）展示了向具有高度贴图地形的场景添加雾（[LU16]的岩石纹理也已应用）。

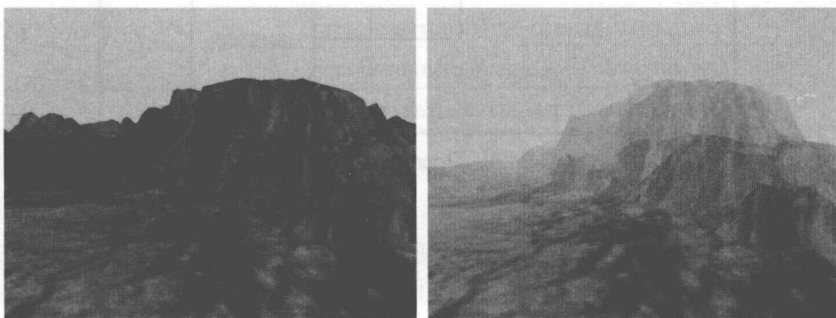


图 14.2 雾的例子

14.2 复合、混合、透明度

我们已经看到了一些混合的例子，比如第 7 章的补充说明以及我们刚才实现的雾。但是，我们还没有看到如何在像素操作期间利用片段着色器之后的混合（或合成）功能（回想一下图 2.2 所示的管线序列）。透明度在那个步骤被处理，我们现在来了解一下。

在本书中，我们经常使用 vec4 数据类型来表示齐次坐标系中的 3D 点和向量。您可能已经注意到我们还经常使用 vec4 来存储颜色信息，其中前 3 个值由红色、绿色和蓝色组成，那么第四个元素是什么？

颜色中的第四个元素称为 Alpha 通道，用来指定颜色的不透明度。不透明度是衡量像素颜色不透明程度的指标。Alpha 值为 0 表示“无不透明度”或完全透明。Alpha 值为 1 表示“不透明度满值”，也就是完全不透明。在某种意义上，颜色的“透明度”是 1-α，其中 α 是 Alpha 通道的值。

回忆一下第 2 章，像素操作利用 Z 缓冲区，当发现另一个对象在该像素的位置更近时，通过替换现有的像素颜色来实现隐藏面消除。我们实际上可以更好地控制这个过程——可以选择混合两个像素。

当渲染一个像素时，它被称为“源”像素。已经在帧缓冲器中的像素（可能是从先前的对象渲染得来）被称为“目标”像素。OpenGL 提供了许多选项，用于决定最终将两个像素中的哪一个或者它们的组合，放置在帧缓冲区中。请注意，像素操作步骤不是可编程阶段——因此用于配置所需合成的 OpenGL 工具可在 C++应用程序中（而不是在着色器中）找到。

用于控制合成的两个 OpenGL 函数是 glBlendEquation(mode)和 glBlendFunc(srcFactor, destFactor)。图 14.3 显示了合成过程的概述。

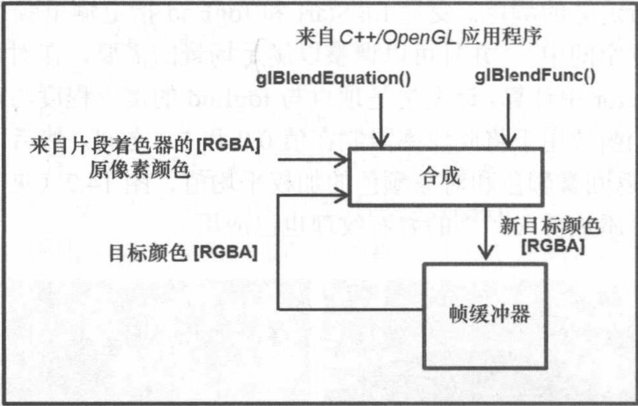


图 14.3 OpenGL 合成概述

合成过程的工作过程如下。

（1）源像素和目标像素分别乘以源因子和目标因子。源和目标因子在 blendFunc()函数调用中指定。

（2）然后使用指定的 blendEquation 来组合修改后的源像素和目标像素以生成新的目标颜色。混合方程在 glBlendEquation()调用中指定。

glBlendFunc()参数的常见选项（即 srcFactor 和 destFactor）如表 14.1 所示。

表 14.1 glBlendFunc()参数的常见选项

glBlendFunc()参数	srcFactor 或 destFactor 的结果
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)

续表

glBlendFunc()参数	srcFactor 或 destFactor 的结果
GL_SRC_COLOR	$(R_{src}, G_{src}, B_{src}, A_{src})$
GL_ONE_MINUS_SRC_COLOR	$(1,1,1,1)-(R_{src}, G_{src}, B_{src}, A_{src})$
GL_DST_COLOR	$(G_{dest}, G_{dest}, B_{dest}, A_{dest})$
GL_ONE_MINUS_DST_COLOR	$(1,1,1,1)-(R_{blendColor}, G_{blendColor}, B_{blendColor}, A_{blendColor})$
GL_SRC_ALPHA	$(A_{src}, A_{src}, A_{src}, A_{src})$
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1, 1)-(A_{src}, A_{src}, A_{src}, A_{src})$
GL_DST_ALPHA	$(A_{dest}, A_{dest}, A_{dest}, A_{dest})$
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1, 1)-(A_{dest}, A_{dest}, A_{dest}, A_{dest})$
GL_CONSTANT_COLOR	$(R_{blendColor}, G_{blendColor}, B_{blendColor}, A_{blendColor})$
GL_ONE_MINUS_CONSTANT_COLOR	$(1,1,1,)-(R_{blendColor}, G_{blendColor}, B_{blendColor}, A_{blendColor})$
GL_CONSTANT_ALPHA	$(A_{blendColor}, A_{blendColor}, A_{blendColor}, A_{blendColor})$
GL_ONE_MINUS_CONSTANT_ALPHA	$(1,1,1,)-(A_{blendColor}, A_{blendColor}, A_{blendColor}, A_{blendColor})$
GL_ALPHA_SATURATE	$(f, f, f, 1)$，其中 $f = \min(A_{src}, 1)$

那些用到“blendColor”（GL_CONSTANT_COLOR 等）的选项需要额外调用 glBlendColor()来指定将用于计算混合函数结果的常量颜色。还有一些其他混合函数未在表 14.1 中显示。

glBlendEquation()参数（混合模式）的可能选项如表 14.2 所示。

表 14.2 glBlendEquation()参数的可能选项

模　式	混合颜色
GL_FUNC_ADD	$result=source_{RGBA}+destination_{RGBA}$
GL_FUNC_SUBTRACT	$result=source_{RGBA}-destination_{RGBA}$
GL_FUNC_REVERST_SUBTRACT	$result=destination_{RGBA}-source_{RGBA}$
GL_MIN	$result=min(source_{RGBA}, destination_{RGBA})$
GL_MAX	$result=max(source_{RGBA}, destination_{RGBA})$

glBlendFunc()默认设置 srcFactor 为 GL_ONE（1.0），destFactor 为 GL_ZERO（0.0）。glBlendEquation()的默认值为 GL_FUNC_ADD。因此，在默认情况下，源像素不变（乘以 1），目标像素被按比例缩小到 0，并且两者相加意味着源像素变为帧缓冲区的颜色。

还有命令 glEnable(GL_BLEND)和 glDisable(GL_BLEND)，它们可用于告诉 OpenGL 应用指定的混合，或忽略它。

我们不会在这里说明所有选项的效果，但我们将介绍一些说明性示例。假设我们在 C++/OpenGL 应用程序中指定以下设置：

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
glBlendEquation(GL_FUNC_ADD)
```

合成将如下进行。

（1）源像素按其 Alpha 值缩放。

（2）目标像素按 1-srcAlpha（源透明度）缩放。

（3）像素值加在一起。

例如，如果源像素为红色，具有 75%不透明度，即[1,0,0,0.75]，并且目标像素包含完全不透明的绿色，即[0,1,0,1]，则结果放在帧缓冲区将是：

```
srcPixel * srcAlpha = [0.75, 0, 0, 0.5625]
destPixel * (1-srcAlpha) = [0, 0.25, 0, 0.25]
resulting pixel = [0.75, 0.25, 0, 0.8125]
```

也就是说，主要是红色，有些是绿色的，而且基本上是实色。这个设置的总体效果是让目标像素以与源像素的透明度相对应的量显示。在此示例中，帧缓冲区中的像素为绿色，输入像素为红色，透明度为 25%（不透明度为 75%）。因此允许一些绿色通过红色显示。

事实证明，混合函数和混合方程的这些设置在许多情况下都能很好地工作。我们将它们应用到包含两个 3D 模型的场景中的实际示例中去：一个环面和环面前的金字塔。图 14.4 显示了这样一个场景，左边是一个不透明的金字塔，右边是金字塔的 Alpha 值设置为 0.8。光照已经添加。

对于许多应用——例如创建平面“窗口”作为房屋模型的一部分，这种简单的透明度实现可能就足够了。但是，在图 14.4 所示的示例中，存在相当明显的不足之处。尽管金字塔模型现在实际上是透明的，但实际透明的金字塔不仅应该显示其背后的对象，还应该显示其自身的背面。

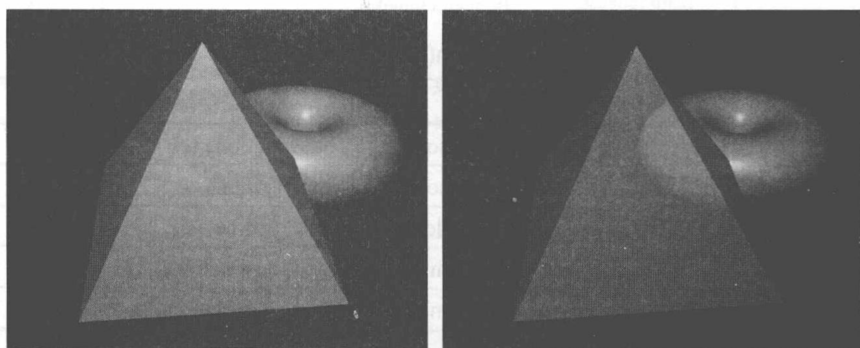


图 14.4　金字塔的 Alpha = 1.0（左），Alpha = 0.8（右）

实际上，金字塔的背面没有出现的原因是因为我们启用了背面剔除。一个合理的想法可能是在绘制金字塔时禁用背面剔除。但是，这通常会产生其他伪影，如图 14.5 左图所示。简单地禁用背面剔除的问题在于混合的效果取决于渲染表面的顺序（因为这决定了源像素和目标像素），并且我们不总是能够控制渲染顺序。通常有利的是首先渲染不透明对象，以及在后面的对象（例如环面），最后再渲染透明对象。这也适用于金字塔的表面，并且在这种情况下，包括金字塔底部的两个三角形看起来不同的原因是它们中的一个在金字塔的前面之前被渲染而一个在之后被渲染。诸如此类的伪影有时被称为“顺序”伪影，并且它们可以在透明模型中显示，因为我们不总是能预测其三角形将被渲染的顺序。

我们可以通过从背面开始分别渲染正面和背面来解决金字塔示例中的问题。程序 14.2 显示了执行此操作的代码。我们通过统一变量来指定金字塔的 Alpha 值并传递给着色器程序，然后通过将指定的 Alpha 替换为计算的输出颜色将其应用于片段着色器中。

另请注意，要使光照正常工作，我们必须在渲染背面时翻转法向量。我们通过向顶点着色器发送一个标志来完成此操作，然后我们在其中翻转法向量。

程序 14.2 透明度的两遍混合

```
C++ / OpenGL 应用程序 —— 在渲染金字塔的 display() 函数中：

. . .
glEnable(GL_CULL_FACE);
. . .
glEnable(GL_BLEND);                  // 配置混合设置
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glBlendEquation(GL_FUNC_ADD);

glCullFace(GL_FRONT);                 // 先渲染金字塔的背面
glProgramUniform1f(renderingProgram, aLoc, 0.3f);          // 背面非常透明
glProgramUniform1f(renderingProgram, fLoc, -1.0f);         // 翻转背面的法向量
glDrawArrays(GL_TRIANGLES, 0, numPyramidVertices);
glCullFace(GL_BACK);                                       // 然后渲染金字塔的正面
glProgramUniform1f(renderingProgram, aLoc, 0.7f);          // 正面略微透明
glProgramUniform1f(renderingProgram, fLoc, 1.0f);          // 正面不需要翻转法向量
glDrawArrays(GL_TRIANGLES, 0, numPyramidVertices);

glDisable(GL_BLEND);

顶点着色器：
. . .
if (flipNormal < 0) varyingNormal = -varyingNormal;
. . .

片段着色器：
. . .
fragColor = globalAmbient * material.ambient + ... etc.      // 和 Blinn-Phong 光照一样
fragColor = vec4(fragColor.xyz, alpha);                      // 使用统一变量中发送的 Alpha 值替换
```

这种“两遍”解决方案的结果如图 14.5 右图所示。

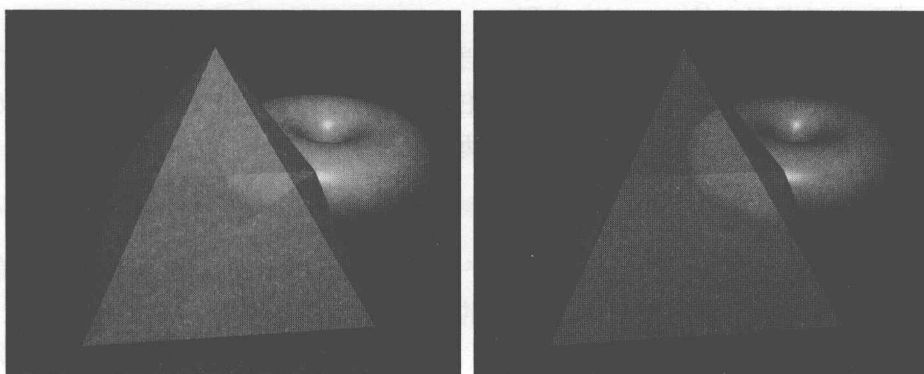


图 14.5 透明度和背面：排序伪影（左）和两遍校正（右）

虽然它在这里运行良好，但程序 14.2 中显示的两遍解决方案并不总是足够的。例如，一些更复杂的模型可能具有面向前方的隐藏表面，并且如果这样的对象变得透明，我们的算法将无法渲染模型的那些隐藏的前向部分。Alec Jacobson 描述了一个适用于大量案例的五遍序列[JA12]。

14.3 用户定义剪裁平面

OpenGL 不仅可以应用于视锥体，还包括了指定剪裁平面的功能。用户定义的剪裁平面的一个用途是对模型切片。这样就可以通过从简单的模型开始并从中切片来创建复杂的形状。

剪裁平面使用平面的标准数学定义来定义：

$$ax + by + cz + d = 0$$

其中 a、b、c 和 d 是用来定义有 X、Y 和 Z 轴的 3D 空间中特定平面的参数。参数表示垂直于平面的向量(a,b,c)，以及从原点到平面的距离 d。可以使用 vec4 在顶点着色器中指定这样的平面，如下所示：

```
vec4 clip_plane = vec4 (0.0,0.0, -1.0,0.2) ;
```

这对应于平面：

$$(0.0)\,x + (0.0)\,y + (-1.0)\,z + 0.2 = 0$$

然后，通过使用内置的 GLSL 变量 gl_ClipDistance[ ]，可以在顶点着色器中实现裁剪，如下例所示：

gl_ClipDistance [0] = dot(clip_plane.xyz, vertPos) + clip_plane.w;

在此示例中，vertPos 指的是在顶点属性（例如来自 VBO）中进入顶点着色器的顶点位置，clip_plane 定义如上。然后我们计算从裁剪平面到传入顶点的带符号距离（如第 3 章所示），如果顶点在平面上，则为 0，或者取决于顶点在平面的哪一侧而为负或正。gl_ClipDistance 数组的下标允许定义多个裁剪距离（即多个平面）。可以定义的最大用户裁剪平面数量取决于图形卡的 OpenGL 实现。

然后必须在 C++/OpenGL 应用程序中启用用户定义的裁剪。内置 OpenGL 标识符 GL_CLIP_DISTANCE0、GL_CLIP_DISTANCE1 等，对应于每个 gl_ClipDistance[ ]数组元素。例如，启用第 0 个用户定义剪裁平面，如下所示。

```
glEnable(GL_CLIP_DISTANCE0);
```

将前面的步骤应用到我们的发光环面会产生如图 14.6 所示的输出，其中环面的前半部分已经被剪裁了（还应用了旋转以提供更清晰的视图）。

可能看起来好像环面的底部也被修剪了，但这是因为环面的内表面没有被渲染。当裁剪会显示形状的内部表面时，也就需要渲染它们，否则模型将显示得不完整（如图 14.6 所示）。

渲染内表面需要再次调用 gl_DrawArrays()，并颠倒缠绕顺序。此外，在渲染背向三角形时，必须反转曲面法向量（如上一节所述）。C++应用程序和顶点着色器的相关修改如程序 14.3 所示，输出如图 14.7 所示。

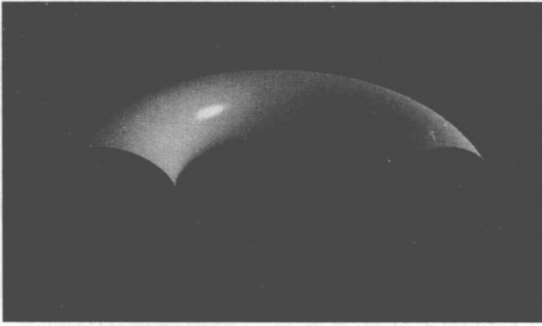


图 14.6　剪裁一个环面

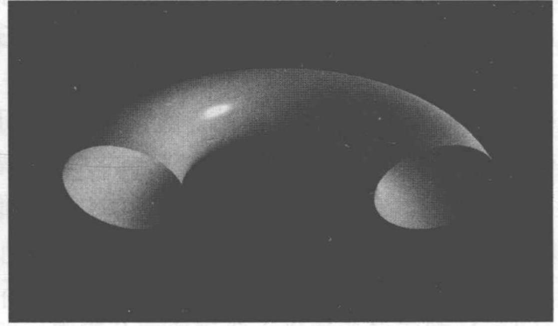


图 14.7　带背面的剪裁

程序 14.3　带背面的剪裁

```
C++ / OpenGL 应用程序：

void display(GLFWwindow* window, double currentTime) {
  . . .
  flipLoc = glGetUniformLocation(renderingProgram, "flipNormal");
  . . .
  glEnable(GL_CLIP_DISTANCE0);

  // 正常绘制外表面
  glUniform1i(flipLoc, 0);
  glFrontFace(GL_CCW);
  glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);

  // 渲染背面，法向量反转
  glUniform1i(flipLoc, 1);
  glFrontFace(GL_CW);
  glDrawElements(GL_TRIANGLES, numTorusIndices, GL_UNSIGNED_INT, 0);
}

顶点着色器：
. . .
vec4 clip_plane = vec4(0.0, 0.0, -1.0, 0.5);
uniform int flipNormal;           // 反转法向量的标志
. . .
void main(void)
{ . . .
  if (flipNormal==1) varyingNormal = -varyingNormal;
  . . .
  gl_ClipDistance[0] = dot(clip_plane.xyz, vertPos) - clip_plane.w;
  . . .
}
```

14.4　3D 纹理

2D 纹理包含由两个变量索引的图像数据，而 3D 纹理包含相同类型的图像数据，但是处在由 3 个变量索引的 3D 结构中。前两个维度仍然代表纹理贴图中的宽度和高度，第三个维度代表深度。

因为 3D 纹理中的数据以与 2D 纹理类似的方式存储，所以很容易将 3D 纹理视为一种

3D“图像”。但是，我们通常不将3D纹理源数据称为3D图像，因为对于这种结构没有常用的图像文件格式（即没有类似的3D版JPEG，至少没有真正三维的图像）。相反，我们建议将3D纹理视为一种物质，我们将其浸没（或“浸入”）被纹理化的对象，从而使对象的表面点从纹理中的相应位置获得颜色。或者可以想象这个物体被从3D纹理“立方体”中“雕刻”出来，就像雕塑家用一块坚固的大理石雕刻出一个人物一样。

OpenGL支持3D纹理对象。为了使用它们，我们需要学习如何构建3D纹理以及如何使用它来纹理化对象。

与可以从标准图像文件构建的2D纹理不同，3D纹理通常是在程序上生成的。正如之前对2D纹理所做的那样，我们决定分辨率，即每个维度中的纹素数量。根据纹理中的颜色，我们可以构建包含这些颜色的三维数组。如果纹理包含可以与各种颜色一起使用的“图案”，我们可能会建立一个保存图案的数组，例如0和1。

例如，我们可以通过填充对应于所需条纹图案的0和1的数组来构建表示水平条纹的3D纹理。假设纹理的所需分辨率是200×200×200纹素，并且纹理由交替的条纹组成，每个条纹高10纹素。通过在嵌套循环中使用适当的0和1填充数组来构建此类结构的简单函数（假设在这种情况下，宽度、高度和深度变量均设置为200）将如下所示。

```
void generate3Dpattern() {
  for (int x=0; x<texWidth; x++) {
    for (int y=0; y<texHeight; y++) {
      for (int z=0; z<texDepth; z++) {
        if ((y/10) % 2 == 0)
          tex3Dpattern[x][y][z] = 0.0;
        else
          tex3Dpattern[x][y][z] = 1.0;
      }
    }
  }
}
```

存储在tex3Dpattern数组中的图案如图14.8所示（见彩插），0呈蓝色，1呈黄色。

使用条纹图案对对象进行纹理处理，如图14.8所示，需要执行以下步骤。

（1）生成如上所示的图案。

（2）使用图案填充所需颜色的字节数组。

（3）将字节数组加载到纹理对象中。

（4）确定对象顶点的适当3D纹理坐标。

（5）在片段着色器中使用适当的采样器来纹理化对象。

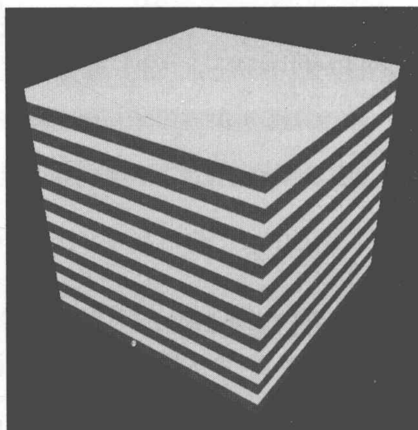


图14.8 条纹3D纹理图案

3D纹理的纹理坐标范围为［0...1］，与2D纹理的方式相同。

有趣的是，步骤（4）（确定3D纹理坐标）通常比最初怀疑的要简单得多。事实上，它通常比2D纹理更简单！这是因为（在2D纹理的情况下）3D对象被2D图像纹理化，我们需要决定如何“展平”3D对象的顶点（例如通过UV映射）来创建纹理坐标。但是当3D纹理化时，对象

和纹理都具有相同的维度。在大多数情况下，我们希望对象反映纹理图案，就像它被“雕刻”出来一样（或浸入其中）。所以顶点位置本身就是纹理坐标！通常所需的只是应用一些简单的缩放以确保对象的顶点的位置坐标映射到 3D 纹理坐标的范围[0, 1]。

由于我们通过程序来生成 3D 纹理，所以我们需要一种从生成的数据中构造 OpenGL 纹理贴图的方法。将数据加载到纹理中的过程与我们之前在第 5.12 节中看到的类似。在这种情况下，我们用颜色值填充 3D 数组，然后将它们复制到纹理对象中。

程序 14.4 展示出了用于实现所有先前步骤的各种组件，以便使用程序构建的 3D 纹理来纹理化具有蓝色和黄色水平条纹的对象。所需的图案在 generate3Dpattern()函数中构建，该函数将图案存储在名为“tex3Dpattern”的数组中。然后在函数 fillDataArray()中构建“图像”数据，按照图案，该函数使用与 RGB 颜色 *R*、*G*、*B* 和 *A* 相对应的字节数据填充 3D 数组，每个数据在[0, 255]范围内。然后将这些值复制到 load3DTexture()函数中的纹理对象中。

程序 14.4 3D 纹理：条纹图案

```
C++ / OpenGL 应用程序:

. . .
const int texHeight= 200;
const int texWidth = 200;
const int texDepth = 200;
double tex3Dpattern[texWidth][texHeight][texDepth];
. . .

// 按照由 generate3Dpattern()构建的图案，用蓝色、黄色的 RGB 值来填充字节数组
void fillDataArray(GLubyte data[ ]) {
  for (int i=0; i<texWidth; i++) {
    for (int j=0; j<texHeight; j++) {
      for (int k=0; k<texDepth; k++) {
        if (tex3Dpattern[i][j][k] == 1.0) {
          // 黄色
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+0] = (GLubyte) 255; // red
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+1] = (GLubyte) 255; // green
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+2] = (GLubyte) 0;   // blue
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+3] = (GLubyte) 255; // alpha
        }
        else {
          // 蓝色
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+0] = (GLubyte) 0;   // red
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+1] = (GLubyte) 0;   // green
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+2] = (GLubyte) 255; // blue
          data[i*(texWidth*texHeight*4) + j*(texHeight*4)+ k*4+3] = (GLubyte) 255; // alpha
} } } } }
// 构建条纹的 3D 图案
void generate3Dpattern() {
  for (int x=0; x<texWidth; x++) {
    for (int y=0; y<texHeight; y++) {
      for (int z=0; z<texDepth; z++) {
        if ((y/10)%2 == 0)
          tex3Dpattern[x][y][z] = 0.0;
        else
          tex3Dpattern[x][y][z] = 1.0;
} } } }
// 将顺序字节数据数组加载进纹理对象
int load3DTexture() {
  GLuint textureID;
```

```
  GLubyte* data = new GLubyte[texWidth*texHeight*texDepth*4];

  fillDataArray(data);

  glGenTextures(1, &textureID);
  glBindTexture(GL_TEXTURE_3D, textureID);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA8, texWidth, texHeight, texDepth);
  glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, texWidth, texHeight, texDepth,
                  GL_RGBA, GL_UNSIGNED_INT_8_8_8_8_REV, data);
  return textureID;
}

void init(GLFWwindow* window) {
  . . .
  generate3Dpattern();                  // 3D 图案和纹理只加载一次，所以在 init()里作
  stripeTexture = load3DTexture();      // 为 3D 纹理保存整型图案 ID
}

void display(GLFWwindow* window, double currentTime) {
  . . .
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_3D, stripeTexture);
  glDrawArrays(GL_TRIANGLES, 0, numObjVertices);
}
```

顶点着色器

```
. . .
out vec3 originalPosition;                  // 原始模型顶点将被用于纹理坐标
. . .
layout (binding=0) uniform sampler3D s;

void main(void)
{ originalPosition = position;             // 将原始模型坐标传递，用作 3D 纹理坐标
  gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

片段着色器

```
. . .
in vec3 originalPosition;            // 接受原始模型坐标，用作 3D 纹理坐标
out vec4 fragColor;
. . .
layout (binding=0) uniform sampler3D s;

void main(void)
{
  fragColor = texture(s, originalPosition/2.0 + 0.5); // 顶点范围为[-1,+1]，纹理坐标范围为[0,1]
}
```

在 C++/OpenGL 应用程序中，load3Dtexture()函数将生成的数据加载到 3D 纹理中。它不使用 SOIL2 来加载纹理，而是直接进行相关的 OpenGL 调用，其方式类似于前面 5.12 节中所述的方式。图像数据应该被格式化为对应于 RGBA 颜色分量的字节序列。函数 fillDataArray()执行此操作，应用黄色和蓝色的 RGB 值，依据由 generate3Dpattern()函数构建并保存在 tex3Dpattern 数组中的条带图案。另请注意 display()函数中指定了纹理类型 GL_TEXTURE_3D。

由于我们希望将对象的顶点位置用作纹理坐标，我们将它们从顶点着色器传递到片段着色器。片段着色器缩放它们，以便它们按照纹理坐标的标准，被映射到范围[0, 1]。最后，

通过 sampler3D 统一变量访问 3D 纹理，该统一变量采用 3 个参数而不是两个参数。我们使用顶点的原始 X、Y 和 Z 坐标，缩放到正确的范围，以访问纹理。结果如图 14.9 所示（见彩插）。

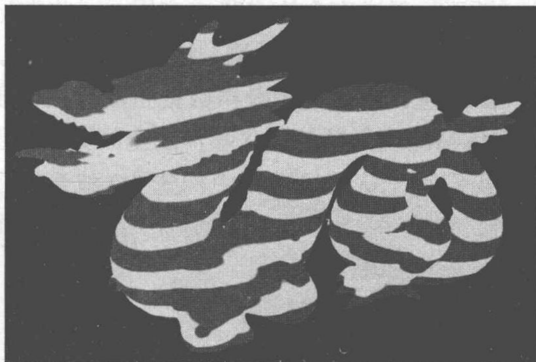


图 14.9　3D 条纹纹理的龙对象

通过修改 generate3Dpattern()可以生成更复杂的图案。图 14.10 显示了将条带图案转换为 3D 棋盘的简单更改，产生的效果如图 14.11 所示。值得注意的是，如果龙的表面采用 2D 棋盘纹理图案进行纹理处理，效果与情况则大不相同。（见习题 14.3。）

```
void generate3Dpattern()
{  int xStep, yStep, zStep, sumSteps;
   for (int x=0; x<texWidth; x++)
   {  for (int y=0; y<texHeight; y++)
      {  for (int z=0; z<texDepth; z++)
         {  xStep = (x / 10) % 2;
            yStep = (y / 10) % 2;
            zStep = (z / 10) % 2;
            sumSteps = xStep + yStep + zStep;
            if ((sumSteps % 2) == 0)
               tex3Dpattern[x][y][z] = 0.0;
            else
               tex3Dpattern[x][y][z] = 1.0;
} } } }
```

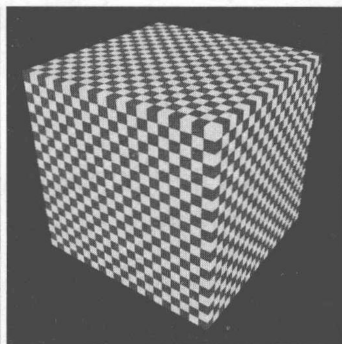


图 14.10　生成棋盘 3D 纹理图案

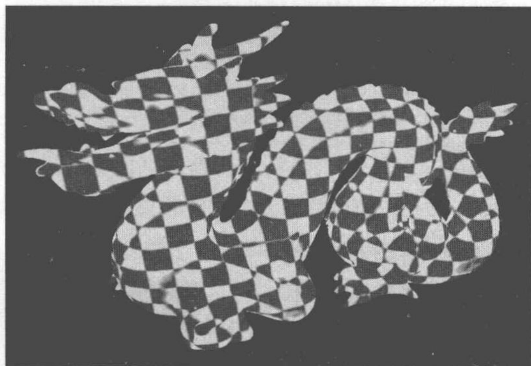


图 14.11　3D 棋盘纹理的龙

14.5　噪声

可以使用随机性或噪声来模拟许多自然现象。一种常见的技术是 Perlin 噪声[PE85]，它以 Ken Perlin 命名。Ken Perlin 在 1997 年因开发生成和使用 2D 和 3D 噪声的实用方法而获得奥斯卡奖。①这里描述的程序基于 Perlin 的方法。

① 由电影艺术与科学学院颁发的奥斯卡技术成就奖。

图形场景中存在许多噪声应用。一些常见的例子是云、地形、木纹、矿产（如大理石中的矿脉）、烟雾、燃烧、火焰、行星表面和随机运动。在本节中，我们将重点关注生成包含噪声的 3D 纹理，然后使用噪声数据生成复杂材质（如大理石和木材），并模拟动画云纹理以用于立方体贴图或天幕。包含噪声的空间数据（例如 2D 或 3D）的集合有时被称为噪声图。

我们首先从随机数据中构建 3D 纹理贴图。这可以使用上一节中显示的函数完成，只需进行一些修改。首先，我们使用以下更简单的 generateNoise()函数替换程序 14.4 中的 generate3Dpattern()函数：

```
#include <random>;
. . .
double noise[noiseWidth][noiseHeight][noiseDepth];
. . .
void generateNoise() {
   for (int x=0; x<noiseWidth; x++) {
       for (int y=0; y<noiseHeight; y++) {
            for (int z=0; z<noiseDepth; z++) {
                noise[x][y][z] = (double) rand() / (RAND_MAX + 1.0);        // 计算出[0...1]范围内的
                                                                            // 一个 double 类型数值
} } } }
```

接下来，修改程序 14.4 中的 fillDataArray()函数，以便将噪声数据复制到字节数组中，以便加载到纹理对象中，如下所示。

```
void fillDataArray(GLubyte data[ ]) {
   for (int i=0; i<noiseWidth; i++) {
        for (int j=0; j<noiseHeight; j++) {
             for (int k=0; k<noiseDepth; k++) {
                  data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] =
                       (GLubyte) (noise[i][j][k] * 255);
                  data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] =
                       (GLubyte) (noise[i][j][k] * 255);
                  data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] =
                       (GLubyte) (noise[i][j][k] * 255);
                  data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] =
                       (GLubyte) 255;
} } } }
```

程序 14.4 的其余部分，用于将数据加载到纹理对象并将其应用于模型，依然保持不变。我们可以通过将它应用于我们的简单立方体模型来查看这个 3D 噪声图，如图 14.12 所示。在此示例中，noiseHeight = noiseWidth = noiseDepth = 256。

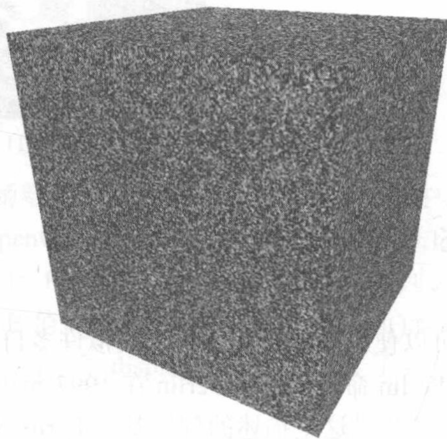


图 14.12 3D 噪声数据纹理的立方体

这是一个 3D 噪声图，虽然它不是非常有用（因为它太嘈杂了，很难有很多实际应用）。为了制作更实用、更可调的噪声模式，我们将使用不同的噪声生成过程替换 fillDataArray()函数。

假设我们使用整数除法作为索引，通过

"放大"，填充数据数组到图 14.12 所示的噪声图的一小部分。对 fillDataArray()函数的修改如下所示。根据用于除法索引的"缩放"因子，可以使得到的 3D 纹理更多或少地呈现"块状"。在图 14.13 中，纹理显示了放大的结果，将索引分别除以缩放因子 8、16 和 32（从左到右）。

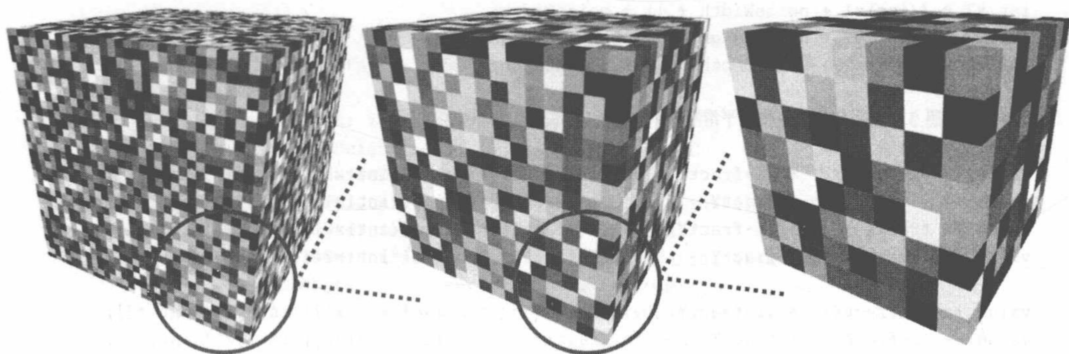


图 14.13 不同"缩放"因子的"块状"3D 噪声图

```
void fillDataArray(GLubyte data[ ]) {
   int zoom = 8;         // 缩放因子
   for (int i=0; i<noiseWidth; i++) {
      for (int j=0; j<noiseHeight; j++) {
         for (int k=0; k<noiseDepth; k++) {
            data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] =
                  (GLubyte) (noise [i/zoom] [j/zoom] [k/zoom] * 255);
            data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] =
                  (GLubyte) (noise [i/zoom] [j/zoom] [k/zoom] * 255);
            data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] =
                  (GLubyte) (noise [i/zoom] [j/zoom] [k/zoom] * 255);
            data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (GLubyte) 255;
} } } }
```

通过从每个离散灰度颜色值插值到下一个灰度颜色值，我们可以平滑特定的噪声图内的"块效应"。也就是说，对于给定 3D 纹理内的每个小"块"，我们通过从其颜色到其相邻块的颜色进行插值来设置块内的每个纹素的颜色。插值代码在下面所示的函数 smoothNoise()中，还有修改后的 fillDataArray()函数。图 14.14 所示的是得到的"平滑"纹理（分别是缩放因子 2、4、8、16、32 和 64——从左到右，从上到下）。请注意，缩放因子现在是一个 double 类型量，因为我们需要小数分量来确定每个纹素的插值灰度值。

```
void fillDataArray(GLubyte data[ ]) {
   double zoom = 32.0;
   for (int i=0; i<noiseWidth; i++) {
      for (int j=0; j<noiseHeight; j++) {
         for (int k=0; k<noiseDepth; k++) {
            data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 +0] =
                  (GLubyte) (smoothNoise(i/zoom, j/zoom, k/zoom) * 255);
            data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 +1] =
                  (GLubyte) (smoothNoise(i/zoom, j/zoom, k/zoom) * 255);
            data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 +2] =
                  (GLubyte) (smoothNoise(i/zoom, j/zoom, k/zoom) * 255);
            data[i*(noiseWidth*noiseHeight*4) + j*(noiseHeight*4) + k*4 +3] = (GLubyte) 255;
} } } }
```

```
double smoothNoise(double x1, double y1, double z1) {
   // x1、y1 和 z1 的小数部分（对于当前纹素，从当前块到下一个块的百分比）
   double fractX = x1 - (int) x1;
   double fractY = y1 - (int) y1;
   double fractZ = z1 - (int) z1;

   // 在 X、Y 和 Z 方向上的相邻像素的索引
   int x2 = ((int)x1 + noiseWidth + 1) % noiseWidth;
   int y2 = ((int)y1 + noiseHeight + 1) % noiseHeight;
   int z2 = ((int)z1 + noiseDepth + 1) % noiseDepth;

   // 通过按照 3 个轴方向插值灰度，平滑噪声
   double value = 0.0;
   value += (1-fractX)  * (1-fractY)  * (1-fractZ) * noise[(int)x1][(int)y1][(int)z1];
   value += (1-fractX)  * fractY      * (1-fractZ) * noise[(int)x1][(int)y2][(int)z1];
   value += fractX      * (1-fractY)  * (1-fractZ) * noise[(int)x2][(int)y1][(int)z1];
   value += fractX      * fractY      * (1-fractZ) * noise[(int)x2][(int)y2][(int)z1];

   value += (1-fractX)  * (1-fractY)  * fractZ     * noise[(int)x1][(int)y1][(int)z2];
   value += (1-fractX)  * fractY      * fractZ     * noise[(int)x1][(int)y2][(int)z2];
   value += fractX      * (1-fractY)  * fractZ     * noise[(int)x2][(int)y1][(int)z2];
   value += fractX      * fractY      * fractZ     * noise[(int)x2][(int)y2][(int)z2];
   return value;
}
```

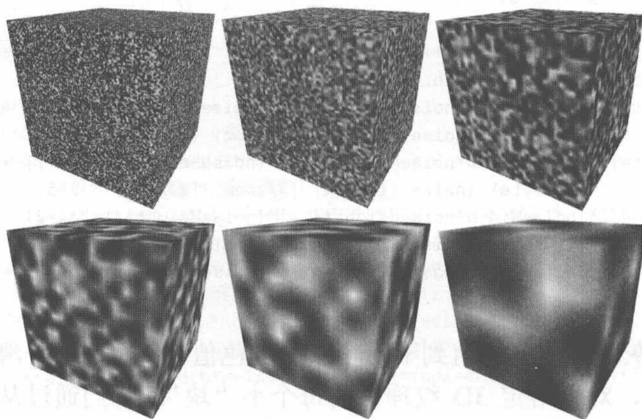


图 14.14 在各种缩放级别平滑 3D 纹理

smoothNoise()函数通过计算相应原始“块状”噪声图中纹素周围的 8 个灰度值的加权平均值来计算给定噪声图的平滑版本中的每个纹素的灰度值。也就是说，它平均纹素所在的小“块”的 8 个顶点处的颜色值。这些“邻居”颜色中的每一个的权重基于纹素与其每个邻居的距离，并归一化到范围[0...1]。

接下来，组合各种缩放因子的平滑噪声图。创建一个新的噪声图，其中每个纹素由另一个加权平均值形成，这次基于每个“平滑”噪声图中相同位置的纹素的总和，其中缩放因子用作权重。这种效应被 Perlin [PE85]称为“湍流”，尽管它与通过求和各种波形产生的谐波实际上更为密切相关。新的 turbulence()函数和 fillDataArray()的修改版本指定了一个噪声图，该图对缩放级别 1～32（2 的各次幂）进行求和，如下所示。其中还显示了以此产生的噪声图在立方体上贴图的结果。

```
double turbulence(double x, double y, double z, double maxZoom) {
   double sum = 0.0, zoom = maxZoom;
```

```
    while (zoom >= 1.0) {                    // 最后一遍是当 zoom = 1 时
        // 计算平滑后的噪声图的加权和
        sum = sum + smoothNoise(x / zoom, y / zoom, z / zoom) * zoom;
        zoom = zoom / 2.0;                   // 对每个 2 的幂的缩放因子
    }
    sum = 128.0 * sum / maxZoom;             // 对不大于 64 的 maxZoom 值，保证 RGB < 256
    return sum;
}

void fillDataArray(GLubyte data[ ] ) {
    double maxZoom = 32.0;
    for (int i=0; i<noiseWidth; i++) {
        for (int j=0; j<noiseHeight; j++) {
            for (int k=0; k<noiseDepth; k++) {
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] =
                        (GLubyte) turbulence(i, j, k, maxZoom);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] =
                        (GLubyte) turbulence(i, j, k, maxZoom);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] =
                        (GLubyte) turbulence(i, j, k, maxZoom);
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] =
                        (GLubyte) 255;
} } } }
```

3D 噪声图（如图 14.15 所示）可用于各种富有想象力的应用。在接下来的部分中，我们将使用它们来生成大理石、木材和云。可以通过放大级别的不同组合来调整噪声的分布。

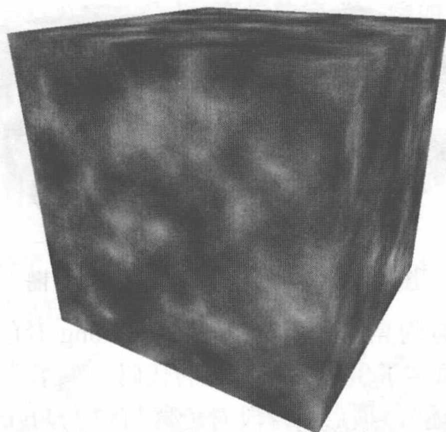


图 14.15 “湍流”噪声的 3D 纹理贴图

14.6 噪声应用——大理石

通过修改噪声图并使用适当的 ADS 材料添加 Phong 照明，我们可以使龙模型看起来像一块大理石般的石头，如图 7.3 所示。

我们首先生成一个条纹图案，有点类似于本章前面的“条纹”示例——新条纹与之前的条纹不同，首先是因为它们是对角线，还因为它们是由正弦波产生的，因此边缘是模糊的。然后，我们使用噪声图来扰动这些线，将它们存储为灰度值。fillDataArray()函数的更改如下：

```
void fillDataArray(GLubyte data[ ]) {
   double veinFrequency = 2.0;
   double turbPower = 1.5;
   double maxZoom = 64.0;
   for (int i=0; i<noiseWidth; i++) {
       for (int j=0; j<noiseHeight; j++) {
           for (int k=0; k<noiseDepth; k++) {
               double xyzValue = (float)i / noiseWidth + (float)j / noiseHeight + (float)k /
                                  noiseDepth + turbPower * turbulence(i,j,k,maxZoom) / 256.0;
               double sineValue = abs(sin(xyzValue * 3.14159 * veinFrequency));

               float redPortion = 255.0f * (float)sineValue;
               float greenPortion = 255.0f * (float)sineValue;
               float bluePortion = 255.0f * (float)sineValue;

               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (GLubyte) redPortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (GLubyte) greenPortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (GLubyte) bluePortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (GLubyte) 255;
} } } }
```

变量 veinFrequency 用于调整条纹数量，turbSize 调整生成湍流时使用的缩放系数，turbPower 调整条纹中的扰动量（将其设置为 0，使条纹不受干扰）。由于相同的正弦波值用于所有 3 个（RGB）颜色分量，所以存储在图像数据阵列中的最终颜色是灰度级的。图 14.16 显示了各种 turbPower 值（0.0、5.5、1.0 和 1.5，从左到右）的结果纹理贴图。

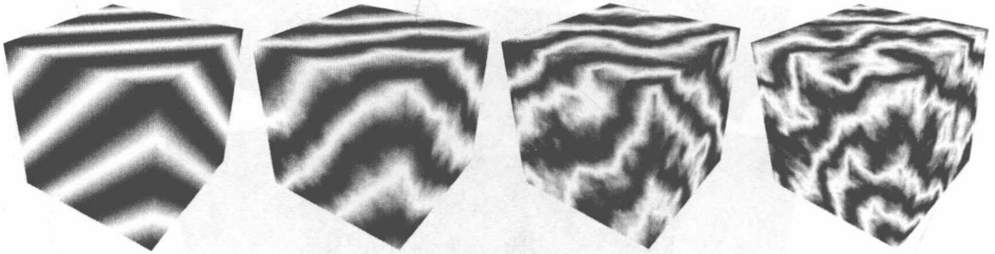


图 14.16 构建 3D“大理石”噪声图

由于我们希望大理石具有闪亮的外观，我们采用 Phong 着色使得“大理石”纹理物体看起来令人信服。程序 14.5 总结了生成大理石龙的代码。除了我们还传递了原始顶点坐标以用作 3D 纹理坐标（如前所述），顶点和片段着色器与用于 Phong 着色的相同。片段着色器使用前面 7.6 节中描述的技术将噪声结果与光照结果结合。

程序 14.5 构建大理石龙

```
C++ / OpenGL 应用程序:

. . .
// 用于 Phong 着色的白光 ADS 设置
float globalAmbient[4] = {0.5f, 0.5f, 0.5f, 1.0f};
float lightAmbient[4] = {0.0f, 0.0f, 0.0f, 1.0f};
float lightDiffuse[4] = {1.0f, 1.0f, 1.0f, 1.0f};
float lightSpecular[4] = {1.0f, 1.0f, 1.0f, 1.0f};
float matShi = 75.0f;

void init(GLFWwindow* window) {
   . . .
   generateNoise();
```

```
    noiseTexture = load3DTexture();             // 和程序 14.4 一样，负责调用 fillDataArray()
}

void fillDataArray(GLubyte data[ ]) {
    double veinFrequency = 1.75;
    double turbPower = 3.0;
    double turbSize = 32.0;
    // 剩下部分构建大理石噪声图的和之前的一样
    . . .
}

顶点着色器
// 和程序 14.4 一样

片段着色器
. . .
void main(void)
{ . . .
    // 模型顶点取值[-1.5, +1.5]，纹理坐标取值[0, 1]
    vec4 texColor = texture(s, originalPosition / 3.0 + 0.5);

    fragColor =
      0.7 * texColor * (globalAmbient + light.ambient + light.diffuse * max(cosTheta,0.0))
        + 0.5 * light.specular * pow(max(cosPhi, 0.0), material.shininess);
}
```

有多种方法可以模拟不同颜色的大理石（或其他石材）。改变大理石中“矿脉”颜色的一种方法是修改 fillDataArray()函数中 Color 变量的定义，例如，通过增加绿色成分：

```
float redPortion = 255.0f * (float)sineValue;
float greenPortion = 255.0f * (float)min(sineValue*1.5 - 0.25, 1.0);
float bluePortion = 255.0f * (float)sineValue;
```

我们还可以引入 ADS 材料值［即在 init()中指定］来模拟完全不同类型的石头，例如“玉石”。

图 14.17（见彩插）显示了 4 个示例，前 3 个使用程序 14.5 所示的设置，第四个示例包含前面图 7.3 所示的“jade” ADS 材料值。

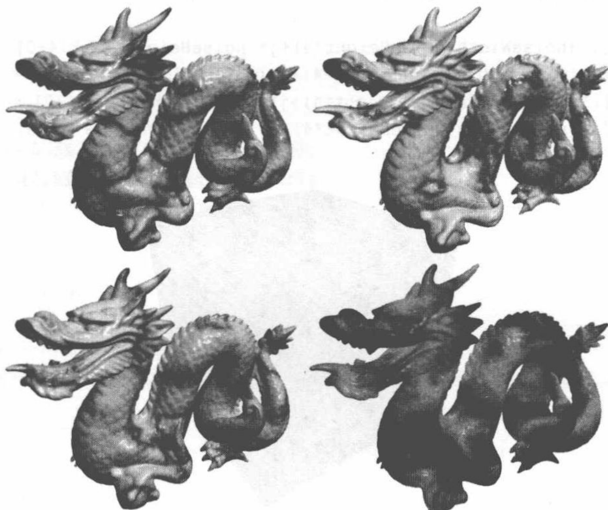


图 14.17　3D 噪声图纹理的龙——3 个大理石和 1 个玉质

14.7 噪声应用——木材

创建“木材”纹理可以采用与之前“大理石”示例中类似的方式。树木按照年轮生长，正是这些年轮成了我们在用木头制成的物体中看到的“木纹”。随着树木的生长，环境压力会在年轮中产生变化，我们也会在木纹中看到这种变化。

我们首先构建一个程序性的“年轮”3D 纹理贴图，类似于本章前面的“棋盘格”。然后，我们使用噪声图来扰动这些年轮，将深色和浅棕色插入年轮纹理贴图中。通过调整年轮的数量以及扰动年轮的程度，我们可以用各种类型的木纹模拟木材。棕色的色调可以通过组合相似数量的红色和绿色、少量蓝色来制作。然后，我们应用具有低“光泽”的 Phong 着色。

我们可以通过修改 fillDataArray()函数来生成环绕我们 3D 纹理贴图中 Z 轴的年轮，使用三角函数指定与 Z 轴等距的 X 和 Y 值。我们使用正弦波循环重复此过程，根据此正弦波均匀地升高和降低红色和绿色成分，以产生不同的棕色调。变量 sineValue 保持精确的色调，可以通过稍微偏移一个分量或另一个分量来调整（在这种情况下，将红色增加 80，将绿色增加 30）。我们可以通过调整 xyPeriod 的值来创建更多（或更少）的年轮。得到的纹理如图 14.18 所示（见彩插）。

```
void fillDataArray(GLubyte data[ ]) {
   double xyPeriod = 40.0;
   for (int i=0; i<noiseWidth; i++) {
       for (int j=0; j<noiseHeight; j++) {
           for (int k=0; k<noiseDepth; k++) {
               double xValue = (i - (double)noiseWidth/2.0) / (double)noiseWidth;
               double yValue = (j - (double)noiseHeight/2.0) / (double)noiseHeight;
               double distanceFromZ = sqrt(xValue * xValue + yValue * yValue)
               double sineValue = 128.0 * abs(sin(2.0 * xyPeriod * distanceFromZ * 3.14159));

               float redPortion = (float)(80 + (int)sineValue);
               float greenPortion = (float)(30 + (int)sineValue);
               float bluePortion = 0.0f;

               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (GLubyte) redPortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (GLubyte) greenPortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (GLubyte) bluePortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (GLubyte) 255;
} } } }
```

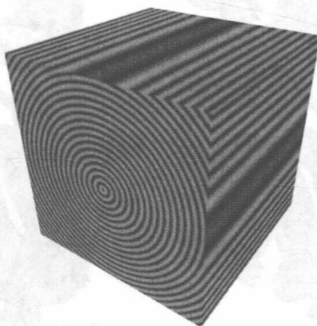


图 14.18 为 3D 木材纹理创建年轮

图 14.18 中的木质年轮环是一个很好的开始，但它们看起来不太逼真——它们太完美了。为了改善这一点，我们使用噪声图（更具体地说，是湍流）来扰动 distanceFromZ 变量，使得环具有轻微的变化。计算修改如下：

```
double distanceFromZ = sqrt(xValue * xValue + yValue * yValue)
                       + turbPower * turbulence(i, j, k, maxZoom) / 256.0;
```

同样，变量 turbPower 调整应用了多少湍流（将其设置为 0.0，产生图 14.18 所示的未受干扰的版本），并且 maxZoom 指定缩放值（在此示例中为 32）。图 14.19 显示了 turbPower 值 0.05、1.0 和 2.0（从左到右）得到的木材纹理。

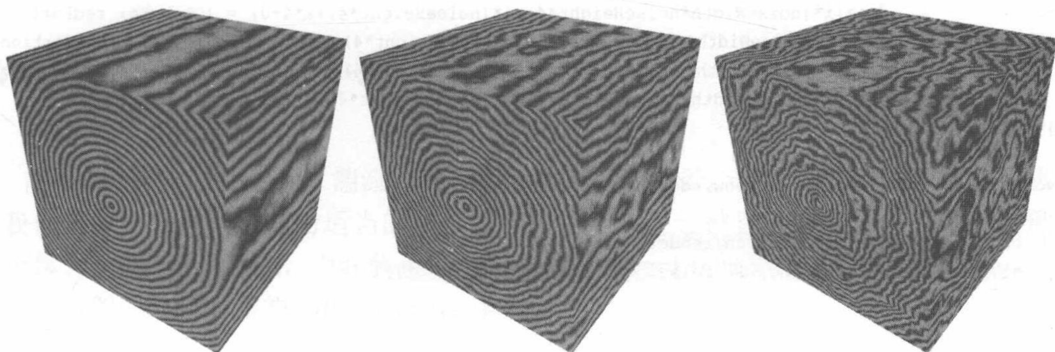


图 14.19 “木材”3D 纹理贴图与噪声地图扰动的年轮

我们现在可以将 3D 木材纹理贴图应用于模型。通过对用于纹理坐标的 originalPosition 顶点位置应用旋转，可以进一步增强纹理的真实感，这是因为用木头雕刻的大多数物品与年轮的方向不完全对齐。为此，我们向着色器发送一个额外的旋转矩阵，以旋转纹理坐标。我们还添加了 Phong 着色，具有适当的木色 ADS 值和适度的光泽度。创建“木质海豚”的完整代码补充和更改见程序 14.6。

程序 14.6 构建木质海豚

```
C++ / OpenGL 应用程序:

glm::mat4 texRot;

// 木质材质（棕色）
float matAmbient[4] = {0.5f, 0.35f, 0.15f, 1.0f};
float matDiffuse[4] = {0.5f, 0.35f, 0.15f, 1.0f};
float matSpecular[4] = {0.5f, 0.35f, 0.15f, 1.0f};
float matShi = 15.0f;

void init(GLFWwindow* window) {
   . . .
   // 旋转应用于纹理坐标——增加额外的木纹变化
   texRot = glm::rotate(glm::mat4(1.0f), toRadians(20.0f), glm::vec3(0.0f, 1.0f, 0.0f));
}

void fillDataArray(GLubyte data[ ]) {
   double xyPeriod = 40.0;
   double turbPower = 0.1;
   double maxZoom = 32.0;
```

```
    for (int i=0; i<noiseWidth; i++) {
        for (int j=0; j<noiseHeight; j++) {
            for (int k=0; k<noiseDepth; k++) {
                double xValue = (i - (double)noiseWidth/2.0) / (double)noiseWidth;
                double yValue = (j - (double)noiseHeight/2.0) / (double)noiseHeight;
                double distanceFromZ = sqrt(xValue * xValue + yValue * yValue)
                                       + turbPower * turbulence(i, j, k, maxZoom) / 256.0;
                double sineValue = 128.0 * abs(sin(2.0 * xyPeriod * distanceFromZ * Math.PI));

                float redPortion = (float)(80 + (int)sineValue);
                float greenPortion = (float)(30 + (int)sineValue);
                float bluePortion = 0.0f;

                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (GLubyte) redPortion;
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (GLubyte) greenPortion;
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (GLubyte) bluePortion;
                data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (GLubyte) 255;
} } } }

void display(GLFWwindow* window, double currentTime) {
   . . .
   tLoc = glGetUniformLocation(renderingProgram, "texRot");
   glUniformMatrix4fv(tLoc, 1, false, glm::value_ptr(texRot));
   . . .
}
```

顶点着色器

```
. . .
uniform mat4 texRot;

void main(void)
{ . . .
   originalPosition = vec3(texRot * vec4(position,1.0)).xyz;
   . . . .
}
```

片段着色器

```
. . .
void main(void)
{ . . .
   uniform mat4 texRot;
. . .
   // 将光照和 3D 纹理结合
   fragColor =
      0.5 * ( . . . )
         +
      0.5 * texture(s,originalPosition / 2.0 + 0.5);
}
```

3D 材质的木质海豚如图 14.20 所示。

片段着色器中还有一个值得注意的细节。由于我们在 3D 纹理内旋转模型，所以有时可能会导致顶点位置因旋转而移动超出所需的[0…1]纹理坐标范围。如果发生这种情况，我们可以通过将原始顶点位置除以更大的数字（例如 4.0 而不是 2.0）来调整这种可能性，然后添加稍大一些的数字（例如 0.6）以使其在纹理空间中居中。

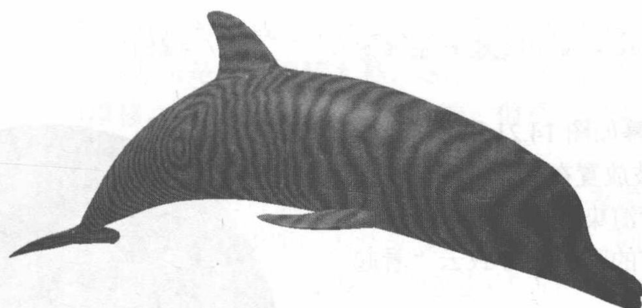


图 14.20 “木材”3D 噪声图纹理的海豚

14.8 噪声应用——云

前面图 14.15 中构建的“湍流”噪声图看起来有点像云。当然，它不是正确的颜色，所以我们首先将它从灰度变为适当的浅蓝色和白色混合。一种直接的方法是为蓝色分量指定一个最大值为 1.0 的颜色，为红色和绿色分量指定 0.0～1.0 的变化（但相等的）值，具体取决于噪声图中的值。新的 fillDataArray()函数如下：

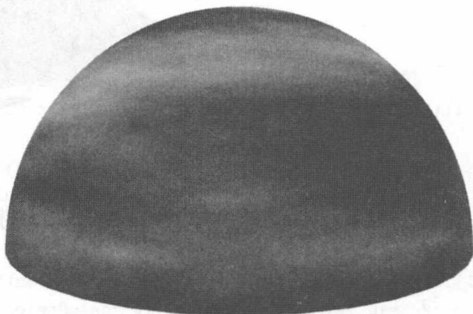
```
void fillDataArray(GLubyte data[ ]) {
   for (int i=0; i<noiseWidth; i++) {
       for (int j=0; j<noiseHeight; j++) {
           for (int k=0; k<noiseDepth; k++) {
               float brightness = 1.0f - (float) turbulence(i,j,k,32) / 256.0f;
               float redPortion = brightness*255.0f;
               float greenPortion = brightness*255.0f;
               float bluePortion = 1.0f*255.0f;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+0] = (GLubyte) redPortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+1] = (GLubyte) greenPortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+2] = (GLubyte) bluePortion;
               data[i*(noiseWidth*noiseHeight*4)+j*(noiseHeight*4)+k*4+3] = (GLubyte) 255;
} } } }
```

生成的蓝色版本的噪声图现在可用于纹理化天幕。回想一下，天幕是一个球体或半球体，在禁用深度测试的情况下被纹理化和渲染，并放置使其围绕相机（类似于天空盒）。

构建天幕的一种方法是使用顶点坐标作为纹理坐标，以与我们对其他 3D 纹理相同的方式对其进行纹理化。然而，在这种情况下，事实证明使用天幕的 2D 纹理坐标会产生看起来更像云的图案，因为球面扭曲会略微拉伸纹理贴图。我们可以通过将 GLSL 的 texture()调用中的第三维设置为常量值来从噪声图中获取 2D 切片。假设天幕的纹理坐标已经以标准方式发送到顶点属性中的 OpenGL 管线，下面的片段着色器使用噪声图的 2D 切片对其进行纹理化：

```
#version 430
in vec2 tc;
out vec4 fragColor;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (binding=0) uniform sampler3D s;
```

```
void main(void)
{ fragColor = texture(s,vec3(tc.x, tc.y, 0.5));          // 常量替代了 tc.z
}
```


图 14.21 云雾缭绕纹理的天幕

得到的纹理化天幕如图 14.21 所示（见彩插）。虽然相机通常被放置在天幕内，但我们在外面使用相机进行渲染，因此可以看到圆顶本身的效果。当前的噪声图导致云“看起来模糊不清”。

虽然我们的朦胧云看起来不错，但我们希望能够塑造它们——也就是说，让它们更多或更少朦胧。一种方法是修改 turbulence()函数，使其使用指数（如 logistic 函数），[①]让云看起来更“明显”。修改后的 turbulence()函数以及相关的 logistic()函数如程序 14.7 所示。完整的程序 14.7 还包含前面描述的 smooth()、fillDataArray()和 generateNoise()函数。

程序 14.7 云纹理生成

```
C++ / OpenGL 应用程序：

double turbulence(double x, double y, double z, double size) {
   double value = 0.0, initialSize = size, cloudQuant;
   while(size >= 0.9) {
      value = value + smoothNoise(x/size, y/size, z/size) * size;
      size = size / 2.0;
   }
   cloudQuant = 110.0;              // 可微调的云质量
   value = value / initialSize;
   value = 256.0 * logistic(value * 128.0 - cloudQuant);
   return value;
}

double logistic(double x) {
   double k = 0.2;        // 可微调的云朦胧程度，产生更多或更少的分明的云边界
   return (1.0 / (1.0 + pow(2.718, -k*x)));
}
```


图 14.22 指数云纹理的天幕

Logistic 函数使颜色更倾向于白色或蓝色，而不是介于两者之间的值，从而产生具有更多不同云边界的视觉效果。变量 cloudQuant 调整噪声图中白色（相对于蓝色）的相对量，这反过来导致当应用 logistic 函数时产生更多（或更少）的白色区域（即不同的云）。由此产生的天幕现在具有更明显的云层，如图 14.22 所示（见彩插）。

最后，真正的云不是静态的。为了增强云的真实感，我们应该通过以下方式使它们变得生

① “logistic”（或“sigmoid”）函数具有 S 形曲线，两端都有渐近线。常见的例子是双曲正切函数和 $f(x) = 1/(1+e-x)$。它们有时也被称为“挤压”函数。

动：（a）使它们随着时间的推移而移动或“漂移”；（b）随着它们漂移逐渐改变它们的形状。

使云“漂移”的一种简单方法是缓慢旋转天幕。这不是一个完美的解决方案，因为真实的云往往会沿着直线方向漂移，而不是围绕观察者旋转。但是，如果旋转缓慢且云只是用于装饰场景，则效果可能是足够的。

随着云的漂移，云逐渐变化，起初可能看起来很棘手。然而，考虑到我们用于纹理云的 3D 噪声图，实际上有一种非常简单而聪明的方法来实现这种效果。回想一下，虽然我们为云构建了一个 3D 纹理噪声图，但到目前为止我们只使用了它的一个“切片”，跟天幕的 2D 纹理坐标相交（我们将纹理查找的 *Z* 坐标设置为一个常量值）。到目前为止，3D 纹理的其余部分尚未使用。

我们的技巧是将纹理查找的常量 *Z* 坐标替换为随时间逐渐变化的变量。也就是说，当我们旋转天幕时，我们逐渐增加深度变量，导致纹理查找使用不同的切片。回想一下，当我们构建 3D 纹理贴图时，我们将平滑应用于沿 3 个轴的颜色变化。因此，纹理贴图中的相邻切片非常相似，但略有不同。因此，通过逐渐改变 texture()调用中的 *Z* 值，云的外观将逐渐改变。

代码更改导致云缓慢移动并随时间变化，如程序 14.8 所示。

程序 14.8　动画云纹理

```
C++ / OpenGL 应用程序：

double rotAmt = 0.0;            // 用来让云看起来漂移的 Y 轴旋转量
float depth = 0.01f;            // 3D 噪声图的深度查找，用来使云逐渐变化
. . .
void display(GLFWwindow* window, double currentTime) {
   . . .
   // 逐渐旋转天幕
   mMat = glm::translate(glm::mat4(1.0f), glm::vec3(domeLocX, domeLocY, domeLocZ);
   rotAmt += 0.02;
   mMat = glm::rotate(mMat, rotAmt, glm::vec3(0.0f, 1.0f, 0.0f));

   // 逐渐修改第三个纹理坐标，以使云变化
   dLoc = glGetUniformLocation(program, "d");
   depth += 0.00005f;
   if (depth >= 0.99f) depth = 0.01f;            // 当我们到达纹理贴图的终点时返回开头
   glUniform1f(dLoc, depth);
   . . .
}

片段着色器
#version 430

in vec2 tc;
out vec4 fragColor;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform float d;

layout (binding=0) uniform sampler3D s;

void main(void)
{ fragColor = texture(s, vec3(tc.x, tc.y, d));            // 逐渐改变的"d"替换前面的常量
}
```

虽然我们无法在单个静止图像中显示逐渐改变漂移和动画的云的效果，但图 14.23 显示了 3D 生成云的一系列快照中的这些变化，因为它们从右到左漂移在天幕上，并在漂移时缓慢改变形状。

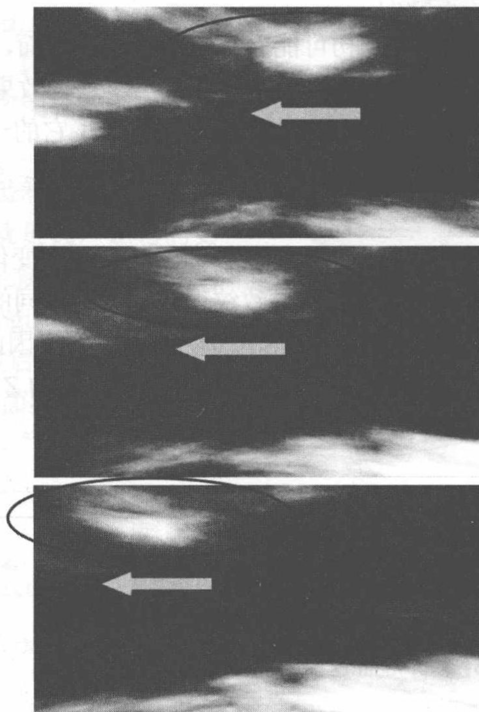


图 14.23 3D 云在漂移时改变

14.9 噪声应用——特殊效果

噪声纹理可用于各种特殊效果。事实上，有许多可能的用途，其适用性仅受到想象力的限制。

我们将在此展示的一个非常简单的特殊效应是*溶解效应*。我们使物体看起来逐渐溶解成小颗粒，直到它最终消失。给定 3D 噪声纹理，可以使用非常少的附加代码实现此效果。

为了促进溶解效果，我们引入了 GLSL 的 discard 命令。此命令仅在片段着色器中是合法的，并且在执行时，它会导致片段着色器丢弃当前片段（意味着不渲染它）。

我们的策略很简单。在 C++/OpenGL 应用程序中，我们创建了一个与图 14.12 所示相同的细粒度噪声纹理贴图，以及随时间逐渐增加的浮点变量计数器。然后，此变量在着色器管线中以统一变量发送，并且噪声图也放置在具有关联采样器的纹理贴图中。然后片段着色器使用采样器访问噪声纹理——在这种情况下，我们使用返回的噪声值来确定是否丢弃该片段。我们通过将灰度噪声值与计数器进行比较来实现这一点，计数器用作一种“阈值”值。因为阈值随着时间的推移逐渐变化，我们可以将其设置为逐渐丢弃越来越多的片段。结果是物体似乎逐渐溶解。程序 14.9 显示了相关的代码部分，它们被添加到程序 6.1 中的

地球渲染球体中。生成的输出如图 14.24 所示。

程序 14.9 使用 discard 命令的溶解效果

```
C++ / OpenGL 应用程序：

float threshold = 0.0f;                    // 用于保留、丢弃片段的逐渐增长的阈值
. . .

在 display() 中
. . .
tLoc = glGetUniformLocation(renderingProgram, "t");
threshold += .002f;
glUniform1f(tLoc, threshold);
. . .
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_3D, noiseTexture);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, earthTexture);
. . .
glDrawArrays(GL_TRIANGLES, 0, numSphereVertices);

片段着色器
#version 430
in vec2 tc;                   // 当前片段的纹理坐标
in vec3 origPos;              // 模型中的原始顶点位置，用于访问 3D 纹理
. . .
layout (binding=0) uniform sampler3D n;          // 用于噪声纹理的采样器
layout (binding=1) uniform sampler2D e;          // 用于地球纹理的采样器
. . .
uniform float t;                 // 用于保留或丢弃片段的阈值
void main(void)
{ float noise = texture(n, origPos).x;           // 从片段中取得噪声值
  if (noise > t)                                 // 如果噪声值大于当前阈值
  { fragColor = texture(e, tc);                  // 则使用地球纹理渲染片段
  }
  else
  { discard;                                     // 否则，丢弃片段（不要渲染）
  }
}
```

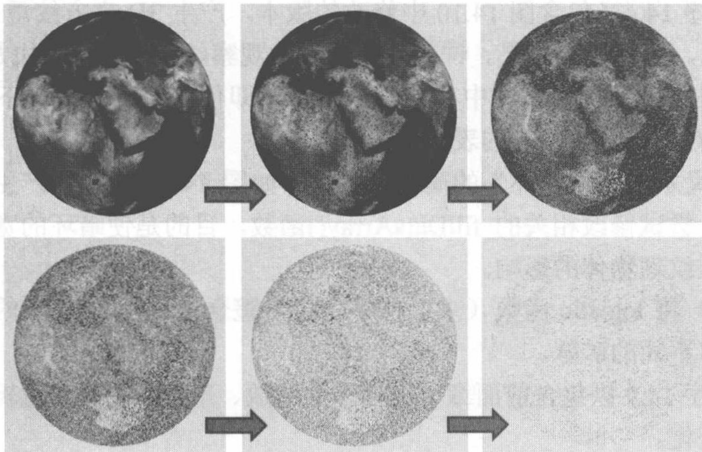


图 14.24 使用 discard 着色器的溶解效果

如果可能，丢弃命令应该谨慎使用，因为它可能会导致性能损失。这是因为它的存在使OpenGL 更难以优化 Z 缓冲深度测试。

补充说明

在本章中，我们使用 Perlin 噪声生成云、模拟木材和大理石般的石头，并且用它们渲染龙。人们发现了 Perlin 噪音的许多其他用途。例如，它可用于创建火焰和烟雾[CC16, AF14]，构建逼真的凹凸贴图[GR05]，并已在电子游戏 Minecraft [PE11]中用于生成地形。

本章生成的噪声图基于 Lode Vandevenne [VA04]描述的程序。我们的 3D 云生成仍存在一些不足之处。纹理不是无缝的，所以在 360° 点有一条明显的垂直线（这也是我们在程序 14.8 中以 0.01 而不是 0.0 开始深度变量的原因，以避免在噪声图的 *Z* 维中遇到接缝）。如果需要，也有用于去除接缝[AS04]的简单方法。另一个问题是在天幕的北峰处，天幕中的球形畸变会产生枕形效应。

我们在本章中实现的云也无法模拟真实云的一些重要方面，例如它们散射太阳光的方式。真正的云也往往在顶部更白，在底部更灰暗。我们的云也没有达到许多实际云所具有的 3D“蓬松”外观。

类似地，存在用于产生雾的更全面的模型，例如 Kilgard 和 Fernando [KF03]描述的模型。

在阅读 OpenGL 文档时，读者可能会注意到 GLSL 包含一些名为 noise1()、noise2()、noise3()和 noise4()的噪声函数，它们被描述为采用输入种子并产生类似高斯的随机输出。我们在本章中没有使用这些函数，因为在撰写本文时，大多数供应商都没有实现它们。例如，无论输入种子如何，许多 NVIDIA 显卡目前只会为这些函数返回 0 值。

习题

14.1 修改程序 14.2 以逐渐增加对象的 Alpha 值，使其逐渐淡出并最终消失。

14.2 修改程序 14.3 以沿水平方向剪裁环面，形成圆形“槽”。

14.3 修改程序 14.4（包含图 14.10 中修改的版本，产生 3D 立方纹理），将它改为纹理化 Studio 522 海豚，然后观察结果。许多人在第一次观察结果时——例如龙上显示的结果，甚至更简单的物体——都认为程序中存在一些错误。即使在简单的情况下，也可以通过从 3D 纹理“雕刻”对象来产生意外的表面图案。

14.4 用于定义木质“年轮环”的简单正弦波（如图 14.18 所示）产生环，其中亮区和暗区的宽度相等。尝试修改相关的 fillDataArray()函数，目的是使暗环的宽度比光环窄。然后观察对所得木质纹理物体的影响。

14.5 （项目）将 logistic 函数（来自程序 14.7）整合进程序 14.5 中的大理石龙，并试验设置以创建更多不同的矿脉。

14.6 修改程序 14.9 以包含前面章节中描述的缩放、平滑、湍流和逻辑步骤。观察所产生的溶解效果的变化。

参考资料

[AF14] S. Abraham and D. Fussell, “Smoke Brush,” Proceedings of the Workshop on Non-Photorealistic Animation and Rendering (NPAR’14), 2014, accessed October 2018.

[AS04] D. Astle, “Simple Clouds Part 1,” gamedev.net, 2004, accessed October 2018.

[CC16] A “Fire Shader in GLSL for your WebGL Games (2016),” Clockwork Chilli (blog), accessed October 2018.

[GR05] S. Green, “Implementing Improved Perlin Noise,” *GPU Gems 2*, NVIDIA, 2005, accessed October 2018.

[JA12] A. Jacobson, “Cheap Tricks for OpenGL Transparency,” 2012, accessed October 2018.

[KF03] M. Kilgard and R. Fernando, “Advanced Topics,” *The Cg Tutorial* (Addison-Wesley, 2003), accessed October 2018.

[LU16] F. Luna, *Introduction to 3D Game Programming with DirectX 12*, 2nd ed. (Mercury Learning, 2016).

[PE11] M. Persson, “Terrain Generation, Part 1,” The Word of Notch (blog), Mar 9, 2011, accessed October 2018.

[PE85] K. Perlin, “An Image Synthesizer,” SIGGRAPH ‘85 Proceedings of the 12th annual conference on computer graphics and interactive techniques (1985).

[VA04] L. Vandevenne, “Texture Generation Using Random Noise,” Lode’s Computer Graphics Tutorial, 2004, accessed October 2018.

附录A　PC（Windows）上的安装与设置

如第 1 章所述，为了在你的计算机上使用 OpenGL 和 C++，必须完成许多安装和设置步骤。这些步骤取决于你所使用的平台。本书中的代码示例运行于 PC（Windows）上；本附录为 Windows 平台提供了逐步设置的详细说明。有关在 Mac 上设置和运行本书中的代码示例的信息，请参阅附录 B。

A.1　安装库和开发环境

A.1.1　安装开发环境

由于我们在本书的整个过程中实现了多个项目，并且在 OpenGL 中有很多库需要协调，所以我们需要以如下方式来设置 C++开发环境，以最大限度地减少每个新建项目所需的配置步骤。这里，我们使用 Visual Studio 2017 [VS17]，类似的步骤也可以应用在其他集成开发环境中。

第一步是在机器上下载并安装 Visual Studio 2017。完成安装后，我们的方法是在单个共享位置安装尽可能多的库，然后创建一个 Visual Studio 的自定义模板，之后，我们创建的每个新项目都已经具有必要的库和依赖项，而不必重新定义。创建模板的描述在附录 A.2.1 中。

A.1.2　安装 OpenGL / GLSL

OpenGL 或 GLSL 并不需要“安装”，但需要确保你的显卡至少支持 OpenGL 4.3 版。如果你不知道机器支持哪种版本的 OpenGL，可以使用各种免费应用程序（例如 GLView [GV16]）来检查。

A.1.3　准备 GLFW

第 1 章中给出了窗口管理库 GLFW 的概述。正如第 1 章中指出的，需要在运行它的机器上编译 GLFW。（请注意，虽然 GLFW 网站包含预编译好的二进制文件下载选项，但它们经常无法正常运行。）编译 GLFW 需要先下载并安装 CMAKE。编译 GLFW 的步骤相对简单。

（1）下载 GLFW 源代码[GF17]。

（2）下载并安装 CMAKE[CM17]。

（3）运行 CMAKE 并输入 GLFW 源代码所在位置和期望的构建目标文件夹。

（4）单击“configure”，如果某些选项以红色高亮，请再次单击“configure”。

（5）单击“generate”。

CMAKE 会在之前指定的“构建”文件夹中生成多个文件。该文件夹中的一个文件名为“GLFW.sln”，这是一个 Visual Studio 项目文件。打开它（当然是使用 Visual Studio）并将 GLFW 编译（构建）为 32 位应用程序（目前比 64 位更稳定）。

生成的构建产生了两个我们需要的项目：

- 由之前的编译步骤生成的 glfw3.lib 文件；
- 原始 GLFW 下载源代码中的“GLFW”文件夹（可在“include”文件夹中找到，它包含我们将使用的两个头文件）。

A.1.4 准备 GLEW

第 1 章我们给出了 GLEW“扩展管理器”库的概述。从 GLEW 官网[GE17]下载 32 位二进制文件。我们需要获得的项目是：

- glew32.lib（在“lib”文件夹中）；
- glew32.dll（在“发布”文件夹中）；
- GL 文件夹，包含多个头文件（在“include”文件夹中）。

A.1.5 准备 GLM

第 1 章给出了数学库 GLM 的概述。访问 GLM 官网[GM17]并下载包含发布说明的最新版本。解压缩后，下载文件夹包含名为“glm”的文件夹。该文件夹（及其内容）是我们需要使用的项目。

A.1.6 准备 SOIL2

第 1 章给出了图像加载库 SOIL2 的概述。安装 SOIL2 [SO17]需要使用一个名为“premake”的工具[PM17]。虽然该过程涉及多个步骤，但它们相对简单。

（1）下载并解压缩“premake”，其中唯一的文件是“premake4.exe”。

（2）下载 SOIL2（使用左侧面板底部的“下载”链接），然后解压缩。

（3）将“premake4.exe”文件复制到 soil2 文件夹中。

（4）打开命令行窗口，导航到 soil2 文件夹，然后输入：

```
premake4 vs2012
```

它应该显示随后创建的文件数量。

（5）在 soil2 文件夹中，打开“make”文件夹，然后打开“windows”文件夹。双击“SOIL2.sln”。

（6）如果 Visual Studio 提示升级库，请单击“确定”按钮。

（7）在右侧面板中，右键单击“soil2-static-lib”并选择“构建（build）”。

（8）关闭 Visual Studio 并导航回 soil2 文件夹。你应该注意到一些新项目。

A.1.7 准备共享的“lib”和“include”文件夹

选择你要存放库文件的位置。你可以随意选择任何文件夹；例如，你可以创建一个文件

夹“C:\ OpenGLtemplate”。在该文件夹中，创建名为“lib”和“include”的子文件夹。

- 在“lib”文件夹中，放置 glew32.lib 和 glfw3.lib。
- 在“include”文件夹中，放置前面描述的 GL、GLFW 和 glm 文件夹。
- 导航回 SOIL2 文件夹，进入其中的“lib”文件夹。将“soil2-debug.lib”文件复制到“lib”文件夹（glew32.lib 和 glfw3.lib 所在的文件夹）。
- 导航回 SOIL2 文件夹，然后导航到“src”。将“SOIL2”文件夹复制到“include”文件夹（GL、GLFW 和 GLM 所在的文件夹）。此 SOIL2 文件夹包含 soil2 的.c 和.h 文件。
- 你可能会发现将“glew32.dll”文件放在此“OpenGLtemplate”文件夹中也很方便，这样你就可以知道在哪里找到它——尽管这不是必要的。

文件夹结构现在应该如图 A.1 所示。

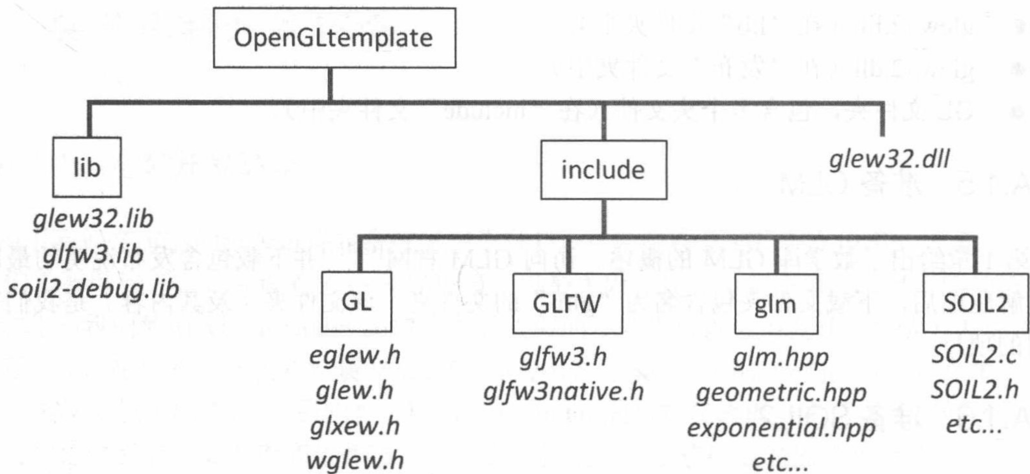


图 A.1　建议的库文件夹结构

A.2　在 MS Visual Studio 中开发和部署 OpenGL 项目

创建 Visual Studio 自定义项目模板

因为我们在 C++ / OpenGL 程序中使用了很多专用库，所以创建 Visual Studio 模板将使启动新的 OpenGL 项目变得更加容易。本节介绍创建和使用此模板的步骤。

启动 Visual Studio（假设为 2017 版本）。创建一个新的空 C++项目。在顶部中心，菜单栏下方有两个相邻的下拉菜单。

- 右边的下拉菜单允许你指定“x86”或“x64”——选择“x86”。
- 左侧的下拉菜单允许你指定是在“调试”模式还是“发布”模式下进行编译。对于这两个模式都需要完成几个步骤。也就是说，它们应该在“调试”模式下完成，然后在“发布”模式下重复。

先在“调试”模式下（然后在“发布”模式下）进入“项目属性”并进行以下更改：

- 在“VC++”下（也可以说是“C / C++”），单击“常规”，然后在“其他包含目录”下添加你之前创建的“include”文件夹。
- 在“链接器”下，有以下两个更改。

  （a）单击“常规”，然后在“其他库目录”下添加先前创建的“lib”文件夹。

  （b）单击“输入”，然后在“附加依赖项”下添加以下 4 个文件名：glfw3.lib，glew32.lib，soil2-debug.lib 和 opengl32.lib（最后一个应该已作为标准 Windows SDK 的一部分提供）。

对“调试”和“发布”模式的项目属性都进行上述更改后，就可以开始创建模板了。创建模板通过进入“项目”菜单并选择“导出模板”来完成。选择“项目”模板，并为模板提供有意义的名称，例如“OpenGL project”。

安装库并设置好自定义模板后，创建一个新的 OpenGL C++项目就很简单了。

（1）启动 Visual Studio，在菜单栏“文件”下选择“新建→项目”。

（2）使用 OpenGL 模板（现在显示为选项）来创建项目。

（3）在右侧的解决方案资源管理器中，在“源文件”下添加“main.cpp”。

（4）在右侧的解决方案资源管理器中，右键单击项目名称，然后在弹出菜单中选择“在文件资源管理器中打开文件夹”。你应该在这里看到 main.cpp 文件。着色器文件也将在开发期间放置在此文件夹中。

（5）向上导航一级到父文件夹，并将“glew32.dll”文件复制到“Release”或“Debug”文件夹中，具体取决于所需的解决方案配置。

在开发、测试和调试应用程序之后，程序可以作为一个单独的可执行文件进行部署。部署时需要在“发布”模式下构建项目，然后将以下文件放在同一个文件夹中。

- 构建项目时生成的.exe 文件。
- 应用程序使用的所有着色器文件。
- 应用程序使用的所有纹理图像和模型文件。
- glew32.dll。

参考资料

[CM17] CMake homepage, accessed December 2017.

[GE17] OpenGL Extension Wrangler (GLEW), accessed December 2017.

[GF17] Graphics Library Framework (GLFW), accessed December 2017.

[GM17] OpenGL Mathematics (GLM), accessed December 2017.

[GV16] GLView, realtech-vr, accessed July 2016.

[PM17] premake homepage, accessed December 2017.

[SO17] Simple OpenGL Image Library 2 (SOIL2), SpartanJ, accessed December 2017.

[SW15] G. Sellers, R. Wright Jr., and N. Haemel, OpenGL SuperBible: Comprehensive Tutorial and Reference, 7th ed. (Addison-Wesley, 2015).

[VS17] Microsoft Visual Studio – downloads, accessed December 2017.

附录B　Macintosh（macOS）平台上的安装与设置

如第 1 章所述，为了在机器上使用 OpenGL 和 C++，必须完成许多安装和设置步骤。这些步骤取决于你希望使用的平台。本附录提供了 Macintosh(macOS)平台逐步设置的详细说明。有关在 Windows PC 上设置和运行本书中代码示例的信息，请参阅附录 A：PC(Windows)上的安装与设置。

在过去的几年中，苹果对 Macintosh(macOS)上 OpenGL 的支持逐渐萎缩。例如，在撰写本文时，现代 Mac 仍然只支持 OpenGL 4.1 版。尽管如此，仍然可以对本书中的示例进行一些修改来运行。在准备必要的库这个步骤中，第 1 章中描述的所有库都是跨平台的，可用于苹果 Macintosh(macOS)。在某些情况下，Mac 上的安装实际上更简单。我们首先介绍如何安装这些库，然后介绍如何配置开发环境。

此外，由于本书中的代码示例用在 Windows 平台上，因此本附录提供了有关转换代码示例的详细信息，以便它们在 Macintosh（macOS）上正确运行。

B.1　安装库和开发环境

B.1.1　准备并安装依赖库

第 1 章概述了每个库的目的和选择。我们不会在此重复这些信息；相反，我们专注于如何安装每个库。

我们首先安装 GLEW 和 GLFW。安装这些库的最简单方法可能是使用“Homebrew”工具。Homebrew 是一个软件包管理器，旨在让用户尽可能简单地在 Mac 上安装常用的实用程序。安装 Homebrew 的步骤如下：

（1）打开 Safari 浏览器，访问 Homebrew 网站。

（2）按照 Homebrew 页面上的安装指引进行操作。具体来说，复制页面中心给出的代码，打开 Mac 上的终端窗口，将复制的命令粘贴到其中，然后点击回车。安装过程中可能需要输入 Mac 密码。

（3）不要关闭终端窗口，在接下来的步骤中我们还会用到它。

接下来，使用新安装的 Homebrew 实用程序来安装 GLEW 和 GLFW，如下。

（1）仍然在终端提示符下输入命令：brew install glfw3。

（2）仍然在终端提示符下输入命令：brew install glew。

（3）请注意，/usr/local/include 路径下现在新增了两个文件夹，分别名为 GL 和 GLFW。

接下来我们安装数学库 GLM。在 4 个库中，它的安装最简单。由于 GLM 是一个仅包含头文件的库，因此只需：（a）按照前面附录 A.1.5 节所述，下载和解压缩库；（b）将生成的“glm”文件夹及其内容复制到/usr/local/include。

安装 SOIL2 可能是安装 4 个库中最棘手的。我们曾使用如下步骤成功安装。

（1）下载 Mac 版本的 SOIL2 和 premake。

（2）解压缩 premake。其中应当只有一个名为 premake 的可执行文件。

（3）将 premake 可执行文件复制到 SOIL2 文件夹中。

（4）在终端窗口中，导航到 SOIL2 文件夹并输入：

```
./premake4 gmake
```

（5）仍然在 SOIL2 文件夹中，输入 cd make/macosx 以导航到 make 文件夹，然后键入：

```
make
```

（6）SOIL2 的构建应该会成功——测试文件可能会构建失败（没关系，它们对我们来说并不重要）。构建会生成“src/SOIL2”文件夹，其中包含几个.h 文件，以及一个“lib”文件夹，文件夹中包含一个名为“libsoil2-debug.a”的库文件。

（7）将包含.h 文件的 SOIL2 文件夹复制到/usr/local/include。

（8）“libsoil2-debug.a”文件可以放在任何能够长期定位的位置。

B.1.2 准备开发环境

在撰写本文时，Mac 版 Visual Studio 2017（在 Windows 平台上运行本书程序的说明中使用的开发环境）不支持 C++。①一个名为 Visual Studio Code 的相关产品可以用来开发 C++，但是幸好 Mac 上面有个更常用的 IDE——Xcode。如果你的 Mac 没有安装 Xcode，那么需要先进行安装，安装过程简单而直接（虽然速度很慢）[XC18]。你可能需要升级操作系统才能安装最新版的 Xcode。

安装 Xcode 之后，你需要配置使其使用 OpenGL 以及上述库。以下是我们为 C++/ OpenGL 应用程序成功设置 Xcode 的步骤。

（1）运行 Xcode，（在 macOS 标签下）创建一个“command line tool”（命令行工具）类型的项目。将语言设置为 C++。

（2）创建一个默认的 main.cpp，它包含一个简单的“hello world”程序。在 Xcode 编辑器中，使用我们的 C++ / OpenGL 应用程序中的所需 main.cpp 代码覆盖该代码。

（3）设置头文件搜索路径，如下所示。

（a）单击项目名称（位于最左侧面板的顶部，蓝色）。

（b）选择主面板顶部中心的“Build Settings”(构建设置)选项卡。

（c）向下滚动到“Search Paths”(搜索路径)部分，确保上方过滤器选择“All”而非“Basic”。

（d）在“Header Search Paths”（头文件搜索路径）中，添加以下路径：/usr/local/include。

① 翻译时的新版 Visual Studio 2019 也不支持 C++。——译者注

（4）将包含“libsoil2-debug.a”文件的文件夹的路径添加到“Library Search Paths”（库搜索路径）。它也位于“Build Settings”中，靠近上一步中使用的头文件搜索路径部分。

（5）为链接阶段设置二进制文件，如下所示。

（a）如有必要，单击项目名称（位于最左侧面板的顶部，蓝色）。

（b）选择主面板顶部中心的“Build Phases”选项卡。

（c）单击“Link Binary with Libraries”旁边的小三角形打开该部分。

（d）在“drag to reorder frameworks”旁边应该有一个“+”，单击“+”。

（e）这里应该打开一个搜索框。搜索“opengl”。应该出现“OpenGL.framework”。选择它并单击“添加”（注意：此 OpenGL 框架已存在于 Mac 中）。

（f）再次单击“+”，这次搜索“core”。应该出现“CoreFoundation Framework”。选择它并单击“添加”（此库也已存在于 Mac 中）。

（g）再次单击“+”，这次单击左下方的“Add Other...”。

（h）浏览窗口打开后，按下 CMD-SHIFT-G。会打开一个输入框；输入/usr/local 并单击“go”。

（i）在显示的文件夹结构中，导航到“Cellar”，然后“glew”，然后导航到显示的版本号为名的文件夹中，然后是“lib”文件夹。其中库文件应以“.dylib”扩展名显示。

（j）选择适当的“.dylib”库文件。它应该命名为：libGLEW.2.1.0.dylib（没有“mx”，也没有快捷方式箭头）。选择后，单击“Open”将其插入。

（k）对 glfw 库重复步骤 g～j。它也在/usr/local/Cellar 中，然后在“glfw”中，它的版本号对应文件夹下的“lib”文件夹中。所需引用的库名为 libglfw.3.2.dylib（没有快捷方式箭头）。单击“Open”将其插入。

（l）对 SOIL2 库文件（我们在 B.1.1 小节中创建的文件）重复该过程。即，单击“+”，单击“Add Other...”，然后导航到放置 libsoil2-debug.a 文件所在的文件夹中。选择该文件，然后单击“Open”将其插入。

（6）设置工作目录，如下所示：单击“Product”菜单的“Edit Scheme”，选中标记为“use custom working directory”的复选框（Xcode10.2.1 中，在 option 选项卡中）。在下方的输入框中，将项目源代码文件夹路径复制进去（包含“main.cpp”文件的文件夹）。

（7）将支持文件（纹理图像、着色器文件和其他支持文件，例如我们在本书中生成的 Utils.cpp 和 Utils.h 文件）复制到“main.cpp”所在的同一工作目录中。

（8）在最左边的面板中，将属于 C++ / OpenGL 应用程序（例如 Utils.cpp，Utils.h，Sphere.cpp 等）的任何其他“.cpp”和“.h”文件添加到项目中，使它们出现在“main.cpp”旁边的左侧面板中。

B.2 修改 Mac 的 C++ / OpenGL / GLSL 应用程序代码

本书中描述的 C++程序中的大部分代码可以直接运行。但是，需要先对少部分代码进行少量修改。大多数更改在“main.cpp”中的“main()”函数中。你可以进行一次修改，并将修改后的 main()函数复制到其他所有项目中。其余部分的更改很小，可以根据需要进行。

这部分描述的变化在研究书中相应的编程部分之前，没有太大意义。读者可以选择跳过本节的部分内容，并在学习相关材料后再回来对照进行修改。虽然可能存在引起混淆的风险，但我们仍然决定在此处对 Macintosh(macOS)平台所有代码进行更改，以便将它们放在同一个地方。

B.2.1 修改 C++代码

让我们从对“main.cpp”文件必需的修改开始。

- Xcode 有时会生成大量的“documentation”警告消息。这会使找到更实质性的错误消息变得更复杂。有几种方法可以阻止这些消息，最简单的一种是将以下两行代码添加到“main.cpp”的顶部：

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wdocumentation"
```

- Homebrew 将 GLEW 安装为 Mac 上的静态库，因此需要在程序顶部，#include <GL/glew.h>命令的正上方添加代码：

```
#define GLEW_STATIC
```

- 在 glfwWindowHint 命令中，将“major”上下文版本设置为 4，将“minor”设置为 1。你需要紧跟着现有的两个 glfwWindowHint 命令，添加另外两个 glfwWindowHint 命令：

```
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

  需要这些命令是因为很多 Mac 默认使用更老的 OpenGL。这些指令会强制使用硬件能够支持的最新 OpenGL 版本。

- 某些 Mac（例如具有视网膜显示屏的 Mac）在设置 GLFW 渲染窗口分辨率时，稍微复杂一些。使用 glfwCreateWindow()创建窗口后，你需要从帧缓冲区中检索实际的屏幕尺寸，如下所示：

```
int actualScreenWidth, actualScreenHeight;
glfwGetFramebufferSize(window, &actualScreenWidth, &actualScreenHeight);
```

  接下来，在 glfwMakeContextCurrent(window)指令后，添加如下代码：

```
glViewport(0,0,actualScreenWidth,actualScreenHeight);
```

  这将确保绘制到帧缓冲区的内容与 GLFW 窗口中显示的内容相匹配。

- 最后，在使用 glewInit()初始化 GLEW 之前，添加如下代码：

```
glewExperimental = GL_TRUE;
```

B.2.2 修改 GLSL 代码

由于 Mac 中使用稍早版本的 OpenGL（V4.1），因此需要对我们的 GLSL 着色器代码（以及一些相关的 C++/OpenGL 代码）中的不同位置进行一些修改：

- 必须修改着色器中指定的版本号。假设你的Mac支持4.1版，则在每个着色器的顶部找到如下代码：

```
#version 430
```

必须将其修改为：

```
#version 410
```

- 4.1版本的OpenGl不支持纹理采样器变量的布局绑定限定符。这会影响从第5章开始的内容。你需要删除布局绑定限定符，并将其替换为另一个完成相同操作的命令。具体来说，在着色器中查找具有以下格式的行：

```
layout (binding=0) uniform sampler2D samp;
```

绑定子句中指定的纹理单元号可能不同（此处为“0”），并且采样器变量的名称可能不同（此处为“samp”）。在任何情况下，你都需要删除布局子句，将它简化为：

```
uniform sampler2D samp;
```

然后，你需要在C++程序中为启用的每个纹理添加如下代码：

```
glUniformli(glGetUniformLocation(renderingProgram, "samp"), 0);
```

这些代码需要紧跟在C++ display()函数中的glBindTexture()命令之后，其中“samp”是统一采样器变量的名称，“0”是先前删除的绑定命令中指定的纹理单元。

B.2.3 补充说明

- 路径名分隔符有时在本书中列为“\”。对于Mac，可能需要将这些更改为“/”。例如：

```
#include <GL\glew.h>
```

可能需要更改为：

```
#include <GL/glew.h>
```

- Macintosh（macOS）必须支持OpenGL 4.1版才能运行本书中的程序。如果你不知道机器支持的OpenGL版本，可查阅苹果网站上提供的列表[AP18]。

参考资料

[AP18] Mac computers that use OpenCL and OpenGL graphics," accessed September 2018.

[GE17] OpenGL Extension Wrangler (GLEW), accessed December 2017.

[GF17] Graphics Library Framework (GLFW), accessed December 2017.

[GM17] OpenGL Mathematics (GLM), accessed December 2017.

[PM17] premake homepage, accessed December 2017.

[SO17] Simple OpenGL Image Library 2 (SOIL2), SpartanJ, accessed December 2017.

[XC18] Apple Developer site for Xcode, accessed January 2018.

附录 C　使用 Nsight 图形调试器

调试 GLSL 着色器代码非常困难。与一般编程语言（如 C++或 Java）中编程不同，在着色器编程中，经常无法确定发生错误的确切位置。通常，着色器错误的表现只是白屏，而不提供有关错误性质的线索。更令人沮丧的是，在运行期间无法打印着色器变量的值，就像平时定位没有头绪的 bug 那样。

我们在 2.2 节列出了一些检测 OpenGL 和 GLSL 错误的技术。尽管这些技术提供了一定的帮助，但缺乏显示着色器变量值这样的基础功能是一个严重的障碍。

出于这个原因，显卡制造商有时会在硬件中提供相关功能，使得可以在着色器运行时从其中提取信息，并提供带图形界面的调试器以访问这些信息的工具。每个制造商的调试工具仅适用于该厂商的显卡。NVIDIA 的图形调试器是 Nsight 工具套件中的一部分，AMD 也有类似的工具套件，名为 CodeXL。本附录介绍如何使用 Nsight。

C.1　关于 NVIDIANsight

Nsight 是 NVIDIA 的一套包含图形调试器的工具套件，它可以在程序运行时查看 OpenGL 图形管道的各个阶段，包括着色器。使用 Nsight 不需要更改或添加任何代码，只要在启用 Nsight 的情况下运行现有程序。Nsight 允许在运行时检查着色器，例如查看着色器的统一变量的当前内容。

Nsight 有适用于 Windows 和 Linux / MacOS 的版本，包括在微软的 Visual Studio（VS）和 Eclipse IDE 下运行的版本。我们将讨论限制在基于 Windows 平台、Visual Studio 版本的 Nsight。（在本书的 Java 版中，我们描述了如何在 Java 程序中使用 VS 版本的 Nsight。）

Nsight 仅适用于兼容的 NVIDIA 显卡，而不适用于 Intel 或 AMD 显卡。NVIDIA 网站[NS18]提供了所支持显卡的完整列表。

C.2　设置 Nsight

设置 Nsight 的过程简单得出奇。以下步骤基于 Nsight 版本 5.3。

（1）如果尚未安装 Visual Studio(VS)，请安装，例如 Visual Studio 2017 社区版。请确保安装组件中包含 C++核心编译器。请注意，Visual Studio 安装可能非常慢。Visual Studio 2017 可在官网上找到。

附录 A 中给出了安装 Visual Studio 的详细步骤。

（2）安装 NVIDIA Nsight，Visual Studio Edition。这应该比在步骤 1 中安装 Visual Studio

的速度快。安装时，不需要 CUDA 元素（除非你出于其他原因需要它们）。

（3）运行 Visual Studio，并确保 Nsight 菜单显示在菜单栏的顶部。

（4）如果尚未加载要运行的程序，请加载该程序。附录 A 给出了配置 C++ / OpenGL 项目的步骤。

C.3　在 Nsight 中运行 C++/OpenGL 应用程序

（1）在“Nsight”菜单下（沿顶部菜单栏），选择“Start Graphics Debugging”，如图 C.1 所示。

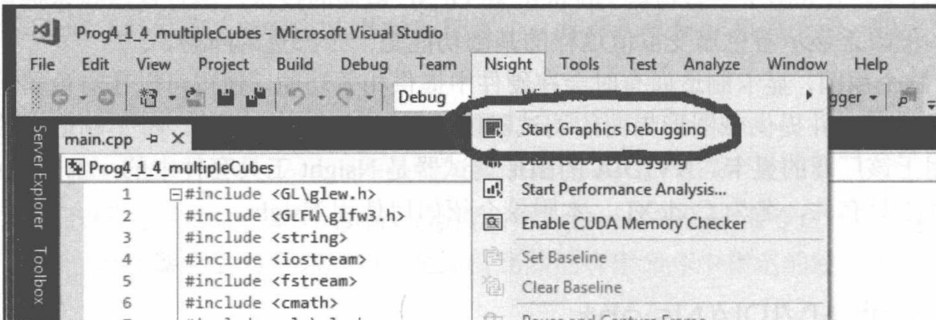


图 C.1　选择“Start Graphics Debugging”

（2）单击后，将弹出一个窗口，询问您是否要“connect without security？”（无安全地连接？），单击“Connect unsecurely”（不安全连接）。这将会启动 C++ / OpenGL 图形程序。你将会看到终端窗口和正在运行的程序。Nsight 可能会在运行的程序中叠加一些信息，如图 C.2 所示。

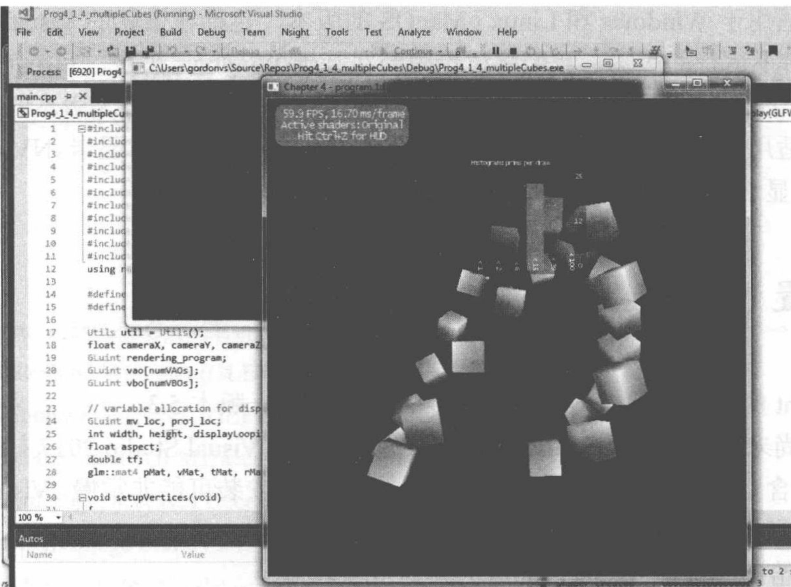


图 C.2　启动 C++ / OpenGL 图形程序

（3）当程序开始运行后，与任意希望检查的部分互动，之后，在 Nsight 菜单中，单击“Pause and Capture Frame”（暂停并捕获当前帧），如图 C.3 所示。

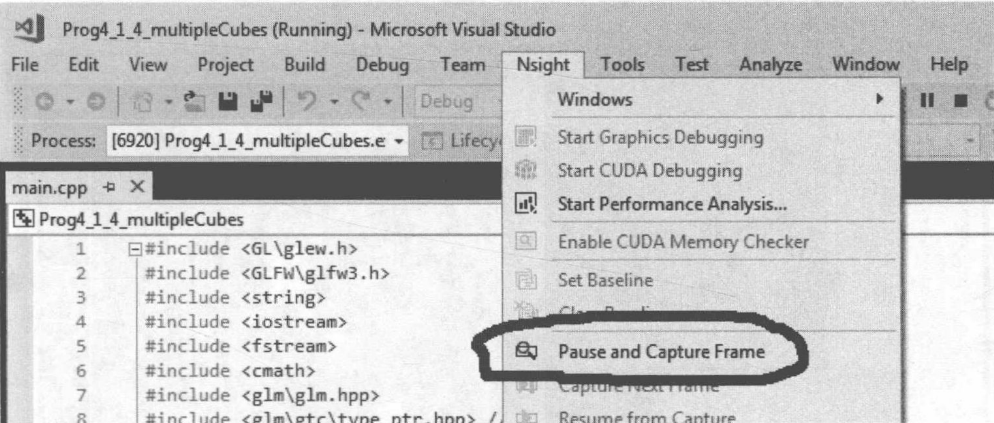


图 C.3　暂停并捕获当前帧

（4）接下来帧调试器界面会出现，同时出现的还有一个 HUD 工具栏和称为“scrubber”的水平选择工具。此程序在这里应该会暂停。在调试器屏幕的核心是左边的工具栏，其中有每个着色器阶段对应的按钮。例如，你可以单击高亮“VS”（顶点着色器），之后在右侧的界面中，你可以向下滚动并查看统一变量的内容（假设你在上方选择了“API inspector”，API 检查器）。在图 C.4 中，“mv_matrix”右侧的小方框已打开，显示 4×4 MV 矩阵的内容。

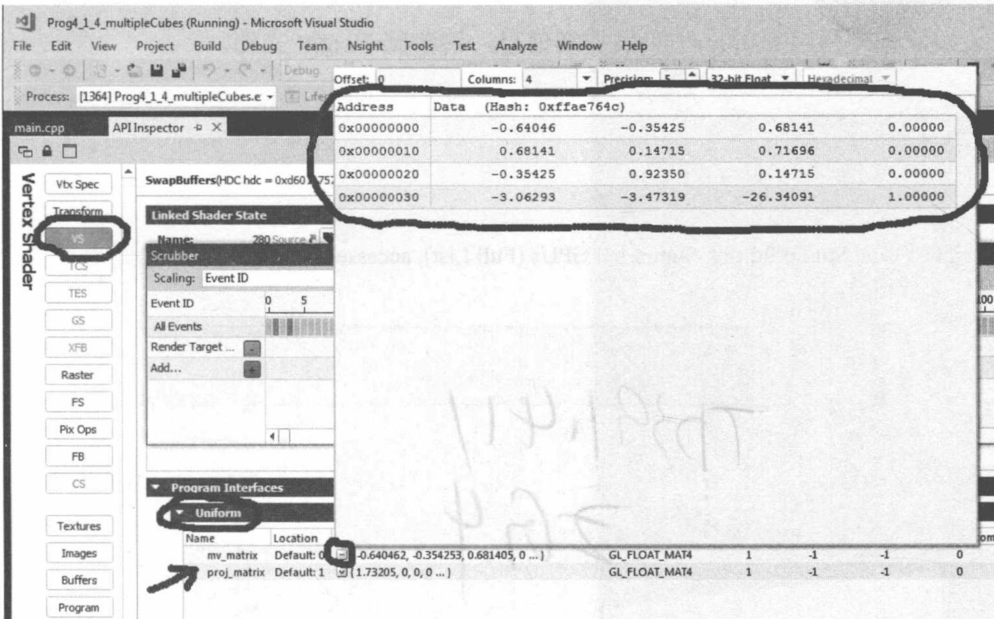


图 C.4　显示 4×4 MV 矩阵

（5）出现的另一个有趣的窗口看起来类似于正在运行的程序。此窗口底部有一个时间轴，你可以单击并查看当前帧中绘制的项目的顺序。图 C.5 是一个示例——注意在时间轴的左侧

区域单击光标，窗口中显示了那些到该时间点为止已经绘制出的项目。

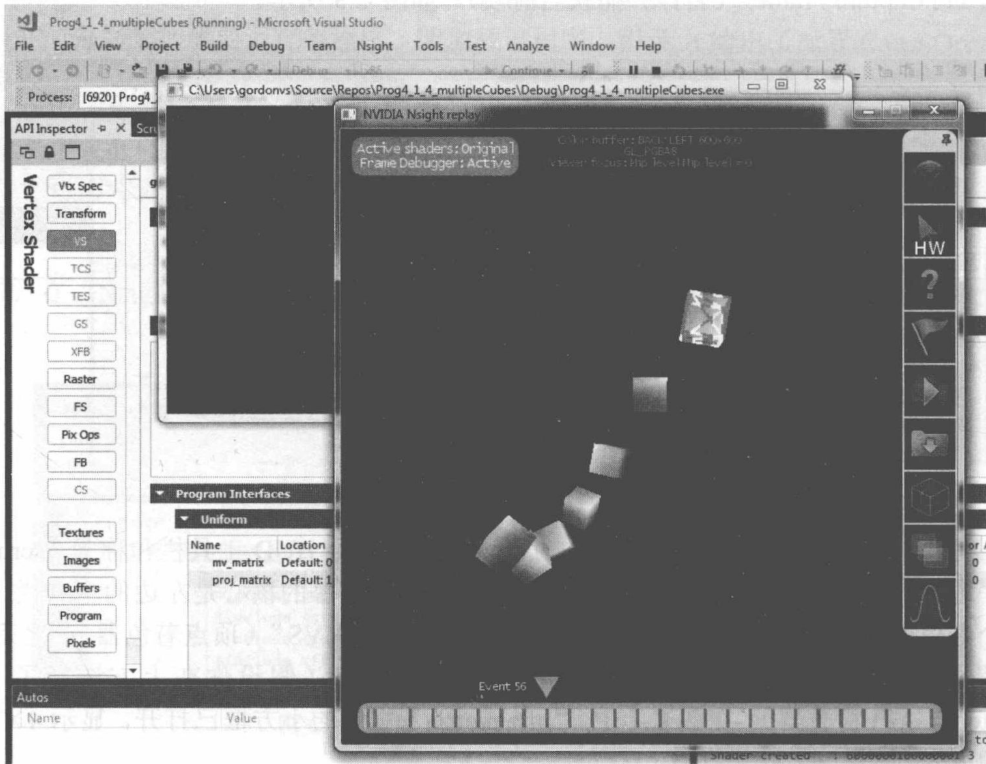


图 C.5 当前帧中绘制的项目顺序

有关如何充分利用 Nsight 工具的详细信息，请参阅 Nsight 文档。

参考资料

[NS18] Nsight Visual Studio Edition Supported GPUs (Full List), accessed May 2018.